Proyecto de Taller V (2000)

Programación de JavaCards

Tutor

Gustavo Betarte

Grupo de trabajo

Daniel Perovich
Leonardo Rodríguez
Martín Varela

Marzo 22, de 2001

Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Resumen

En este documento se reporta el trabajo desarrollado en el contexto de un Taller V cuyo objetivo principal es el estudio de la tecnología JavaCard. Se ha realizado un estado del arte de esta tecnología, profundizando, en particular, en un tema de interés actual como lo es el uso compartido de objetos en forma segura. Asimismo, el presente informe presenta y discute las diversas herramientas de desarrollo con las que se ha experimentado para la programación de aplicaciones JavaCard. En particular se introduce una herramienta que ha sido desarrollada por este equipo de trabajo para facilitar la programación de aplicaciones. Se presenta, finalmente, el caso de estudio que se ha desarrollado, el cual consiste en la historia clínica resumida del portador de la tarjeta.

Tabla de contenido

Introducción

Numerosos campos de aplicación de la Internet y de las tarjetas inteligentes - especialmente el comercio electrónico - requieren que datos y recursos sean protegidos por medio de mecanismos de seguridad sofisticados. El lenguaje de programación Java representa una respuesta práctica a las cuestiones de movilidad y seguridad sobre Internet. Actualmente, se puede afirmar que Java se ha impuesto como un estándar de facto en estos dominios de aplicación donde las exigencias de seguridad son muy altas.

Recientemente, Sun Microsystems Inc. publicó la definición de un nuevo miembro de las tecnologías Java, llamada JavaCard, que está orientada a la programación de tarjetas inteligentes. JavaCard fue diseñado de tal forma que ciertas construcciones de Java consideradas como demasiado complejas o no aplicables para la programación de tarjetas inteligentes no son incorporadas y por otro lado se agregan facilidades específicas para el manejo de transacciones con tarjetas inteligentes (atomicidad de un grupo de operaciones, objetos persistentes, etc.). Las políticas de seguridad de Java (conocidas como el sandbox model) que prohiben cualquier interacción entre objetos de diferentes applets fueron modificadas, y en algunos casos, debilitadas: JavaCard, por ejemplo, permite que un objeto sea compartido por diferentes applets.

Este documento presenta un estado del arte de la tecnología JavaCard, definiendo los conceptos de SmartCard y JavaCard. Se describe, además, las características principales de la especificación JavaCard 2.1 y se introduce las herramientas de desarrollo con las que se ha trabajado, el emulador de Sun y el kit de desarrollo Cyberflex Access de Schlumberger.

Se desarrolló una herramienta, *proxy*, motivada en un principio por las dificultades que presenta el trabajar con el emulador, y diseñada en busca de la simplificación del desarrollo de aplicaciones. También se explica en detalle el mecanismo de objetos compartidos introducidos en la versión 2.1 de la especificación, y se plantea los problemas que ésta presenta. Se propone, como mecanismo para la resolución de dichos problemas, una metodología de desarrollo para compartir objetos en forma segura.

El caso de estudio que se ha considerado consiste en un sistema para mantener la historia clínica de un paciente en una tarjeta JavaCard. Dadas las características del problema a resolver, se optó por un enfoque más general, diseñando una solución que permite atacar problemas en donde la estructura de la información pueda ser representada por una estructura arborescente con información en las hojas. Se utilizó SML (Simple Markup Language) para el formato de los archivos de entrada y salida ya que éste brinda una estructura legible.

El presente Taller V se inscribe dentro de las líneas de investigación y desarrollo del Grupo de Métodos Formales del In.Co. y forma parte de un proyecto más ambicioso de dicho grupo cuyo objetivo es la especificación y verificación de políticas de seguridad de programas Java y el desarrollo de herramientas para la generación de aplicaciones confiables en el contexto JavaCard. Un objetivo adicional considerado en este trabajo ha sido el diseño y desarrollo de una aplicación de porte medio para tarjetas inteligentes.

Estructura del documento

La primer sección brinda una introducción al tema tratado y presenta el estado del arte desarrollado en el proyecto, indicando aplicaciones en las que está siendo utilizada la tecnología estudiada. En la segunda sección se comenta las herramientas de desarrollo con las que se ha trabajado en el proyecto y se introduce la herramienta *proxy*. La metodología de desarrollo para compartir objetos en forma segura es presentada en la tercer sección. El caso

de estudio desarrollado se presenta en la cuarta sección, junto con una discusión del problema planteado y otras posibles soluciones al mismo. La quinta sección presenta las conclusiones obtenidas a lo largo del trabajo realizado, y algunas sugerencias para futuros trabajos en este contexto. En el Apéndice I se incluye un artículo de divulgación escrito como parte del presente trabajo, y que trata con mayor profundidad la metodología presentada en la sección 3. Un ejemplo de utilización de la herramienta *proxy* es presentado en el Apéndice II. El ejemplo utilizado es una versión simplificada de un monedero electrónico. El Apéndice III presenta los detalles del caso de estudio, como ser la información a almacenar y los archivos de definición de los datos, entre otros. Por último, el Apéndice IV define los conceptos básicos de SML, con el fin de brindar una referencia rápida al lector.

Sección 1
Estado del Arte

1.1. Contexto general

1.1.1. SmartCard

Una SmartCard es un dispositivo del tamaño de una tarjeta de crédito[1], el cual almacena y procesa información mediante un circuito de silicio embebido en el plástico de la tarjeta. [ZHI1]

Las tarjetas magnéticas (Magnetic Stripe Cards) utilizadas en cajeros automáticos y como tarjetas de crédito, son antecesoras de las SmartCards. Las mismas almacenan información en una banda magnética de tres pistas que llevan adherida sobre la superficie.

Las SmartCards han existido en varias formas desde 1974. Desde ese entonces, gracias a la motivación de compañías como Gemplus [GEM] y Schlumberger [SLB1], han recibido gran atención en el mercado de los dispositivos de control. Según la consultora Frost & Sullivan [FROST] más de 600 millones de SmartCards fueron emitidas en 1996 y se espera un consumo de 21 billones para el año 2010 [CHAN].

Puede clasificarse a las SmartCards en dos tipos:
- Memory Cards (también llamadas Low-End): Tienen circuitos de memoria que permiten almacenar datos. Estas tarjetas utilizan cierta lógica de seguridad, a nivel del hardware, para controlar el acceso a la información.
- Microprocessor Cards (también llamadas High-End): Tienen un microprocesador que les brinda una limitada capacidad de procesamiento de datos. Tiene capacidad de lectura, escritura y procesamiento.

Las SmartCards contienen tres tipos de memoria: ROM, EEPROM y RAM. Normalmente no contienen fuente de poder, display o teclado. Interactúa con el mundo exterior utilizando una interfaz serial con ocho puntos de contacto[2] como se muestra en la Figura 1.1. Las dimensiones y ubicación de los mismos están especificados en la parte 2 de estándar ISO 7816 [ISO].

[1] Se debe aclarar que la tecnología de SmartCards no se reduce estrictamente a tarjetas, como lo demuestra el amplio uso de los iButtons de Dallas Semiconductor [IBUT1], los cuales se comercializan en varias versiones, incluida una que implementa el JavaCard API 2.0

[2] Excepto las que trabajan con radiofrecuencia. Ver párrafo siguiente

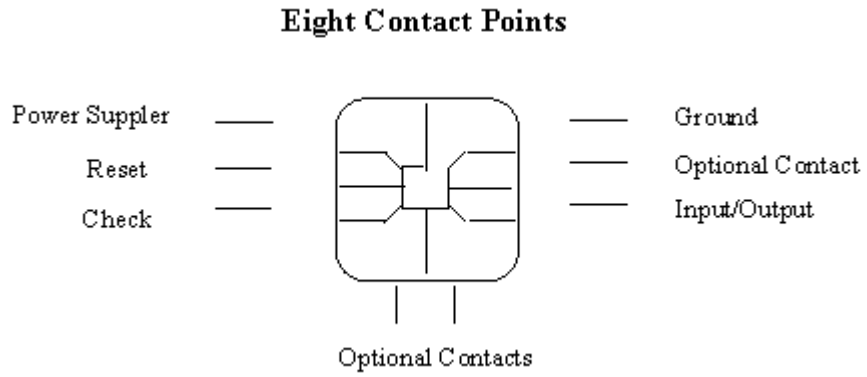


Figura 1.1

Son utilizadas en conjunto con un Card Acceptance Device (CAD o Reader), que puede, aunque no necesariamente, estar conectado a una computadora (Figura 1.2). Este dispositivo tiene como objetivo el proveer la fuente de poder e interfaz de comunicación para las tarjetas. La interacción entre la tarjeta y el CAD puede hacerse mediante un punto de contacto (la tarjeta debe ser insertada en el CAD) o sin contacto (utilizando radiofrecuencia).

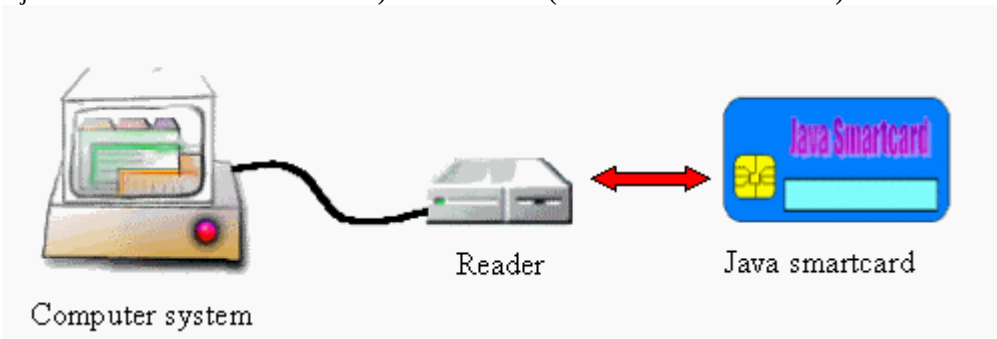


Figura 1.2

1.1.2. JavaCard

Dentro de la categoría de SmartCards con microprocesador se encuentran las llamadas JavaCards o Java SmartCards. Una JavaCard es una SmartCard capaz de ejecutar programas desarrollados en Java. Originalmente, las aplicaciones para SmartCards eran escritas en lenguajes específicos de los proveedores de SmartCards (normalmente el ensamblador del microprocesador utilizado, o eventualmente C). El lema "escribe una vez, corre en cualquier lado" hizo que Java fuese una solución a este problema [ZHI1]. La primera JavaCard en salir al mercado fue producida por Schlumberger, aún antes de que Sun fijara el standard. [DIGIOR]

En pocas palabras, una JavaCard es una tarjeta con microprocesador que puede ejecutar programas (llamados applets) escritos en un subset[3] del lenguaje Java.

Los componentes principales dentro de una JavaCard son el microprocesador y las memorias. La arquitectura básica de una JavaCard consiste de Applets, JavaCard API, JavaCard Virtual Machine (JCVM) [SUN2] / JavaCard Runtime Environment (JCRE) [SUN1], y el sistema operativo nativo de la tarjeta (Figura 1.3).

[3] Estrictamente hablando, este lenguaje no es exactamente un subset de Java, ya que introduce algunos elementos nuevos en la JVM, como ser mecanismos de seguridad adicionales.

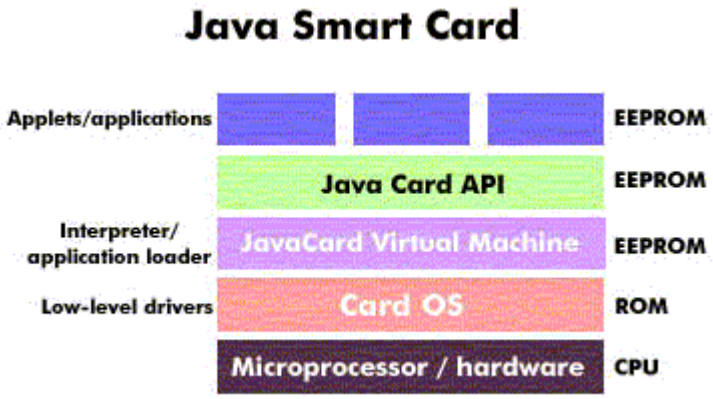


Figura 1.3

La máquina virtual corre sobre el sistema operativo de la tarjeta, y se puede ver como una abstracción del mismo. Sobre la máquina virtual se encuentra el JavaCard Framework, que es un conjunto de clases necesarias para el funcionamiento del JCRE y JavaCard API, así como clases utilitarias comerciales o extensiones propietarias del fabricante de la tarjeta. Finalmente, haciendo uso de este ambiente se encuentran los applets. [CHAN, ZHI1]

1.1.3. Aplicaciones de JavaCard

El número de aplicaciones para JavaCards, y SmartCards en general, va en un aumento constante, y abarca áreas muy diversas. Algunos ejemplos típicos se citan a continuación:

- *Electronic Purse o Electronic Wallet (ePurse y eWallet)*: esta aplicación se utiliza como dinero electrónico. Se puede fijar un monto de dinero inicial, sobre el cual se puede realizar operaciones de débito, crédito o consulta, y puede ser utilizado para el pago o cobro de servicios o bienes. Típicamente lleva asociado algún sistema de seguridad (por ejemplo un PIN), para evitar la posibilidad de fraude.

- *Transacciones Seguras:* Ya sea a través de cajeros automáticos o de Internet, las SmartCards proveen un nivel de seguridad muy superior al de las tarjetas magnéticas comunes o los sistemas basados en contraseñas o cookies, ya que es normal que incluyan un API de criptografía fuerte.

- *Identificación Digital / Firma Digital*: este tipo de aplicaciones se utiliza para validar la identidad del portador de la tarjeta, o para poder certificar el origen de ciertos datos. Normalmente se basan fuertemente en las primitivas criptográficas del API y/o las que están implementadas en hardware.

- *Programas de Lealtad*: Este tipo de aplicación sirve a las empresas que ofrecen servicios preferenciales para clientes frecuentes para poder validar la identidad del cliente, y para descentralizar la información. Suponiendo que se tiene un sistema de puntos acumulables, en el cual participan varias empresas, esto simplifica mucho el tratamiento de los datos, evitando tener que compartir una gran base de datos o tener que realizar réplicas de las distintas bases.

- *Sistemas de Prepago*: En estos sistemas, un cliente "carga" su tarjeta con una cierta "cantidad" de servicio, la cual va siendo decrementada a medida que el cliente hace uso del servicio. El servicio puede variar desde telefonía celular hasta TV cable, pasando por acceso a sitios web o transporte público.

- *Health Cards*: En algunos hospitales ya se está implementando un sistema de identificación de pacientes y almacenamiento de los principales datos de la historia clínica de los mismos en SmartCards para agilitar la atención. Actualmente la capacidad de almacenamiento es muy limitada, pero en un futuro se podría almacenar toda la historia

clínica (incluidas radiografías y similares) en una SmartCard. La sección 4 presenta el caso de estudio desarrollado; el mismo consta de la implementación de una Health Card.

- *Control de Acceso y de Asistencia*: Ya hay varios sistemas de control de acceso y asistencia implementados en base a SmartCards o a iButtons. [CHAN, SUN2]

Existen más aplicaciones de SmartCards y JavaCards, algunas son variantes de las mencionadas arriba, mientras otras satisfacen necesidades específicas de una empresa.

1.2. JavaCard API 2.0 y 2.1

La última versión de la plataforma JavaCard es la especificación 2.1.1, liberada por Sun en el 2000, y actualmente hay algunas implementaciones en el mercado o en vías de ser comercializadas [GEM, SLB1].

Existen además otros sistemas basados en la especificación 2.0, como los iButtons [IBUT1, IBUT2], que ya están ampliamente difundidos.

La principal innovación que se introdujo en la versión 2.1 es la facilidad de compartir objetos entre applets, tema tratado con mayor profundidad en la las subsección 1.5, en la Sección 3 y en el Apéndice I.

1.3. Funcionamiento de una JavaCard

1.3.1. Introducción

La tecnología JavaCard combina parte del lenguaje de programación Java con un entorno de ejecución optimizado para SmartCards y similares. El objetivo de la tecnología JavaCard es llevar los beneficios del desarrollo de software en Java al mundo de las SmartCards. [SUN1]

1.3.2. JavaCard Runtime Environment (JCRE)

El JCRE comprende la máquina virtual de JavaCard (JCVM) junto a las clases y servicios definidos en el Application Programming Interface (API). Sobre este ambiente se ejecutan los applets que se desarrollan.

La JCVM se diferencia principalmente de una JVM normal en que el tiempo de vida de la misma es igual al tiempo de vida de la tarjeta, por lo cual los objetos[4] mantienen sus estados entre dos sesiones con una terminal (CAD). Es responsabilidad del JCRE garantizar este comportamiento. Cuando se retira la tarjeta del CAD, se asume que se está ejecutando en un ciclo de reloj de período infinito [SUN1]. Otras diferencias entre ellas son las limitaciones en los tipos de datos manejados y los requerimientos de hardware para la ejecución [SUN2].

Para poder comprender como funciona una JavaCard, hay que tener en cuenta que al realizar la especificación de la plataforma, Sun se apegó al estándar ISO 7816 [ISO], el cual establece, entre otras cosas, la forma de comunicación entre una SmartCard y un CAD. De acuerdo al ISO 7816, el intercambio de información y comandos entre la tarjeta y el CAD se realiza a través de APDUs (Application Protocol Data Units), los cuales son paquetes de información con un formato específico [SUN1, ZHI1, DIGIOR, ZHI2]. De acuerdo al estándar, las SmartCards nunca inician la comunicación con el CAD, sino que sólo responden a los comandos que éste le envía. Se define dos tipos de APDU, los llamados COMMAND APDU, que son los que envía el CAD a la tarjeta, y los RESPONSE APDU, que son los que envía la tarjeta al CAD como respuesta a un COMMAND APDU.

[4] Salvo aquellos que son marcados como *transient*

La siguiente tabla describe el formato de ambos tipos de APDU.

Command APDU						
Encabezado Obligatorio				Cuerpo Opcional		
CLA	INS	P1	P2	LC	Data field	LE

Response APDU		
Cuerpo Opcional	Cola Obligatoria	
Data field	SW1	SW2

Campo	Tamaño	Descripción
CLA	1 byte	Clase de instrucción. Indica la estructura y el formato para una categoría de COMMAND y RESPONSE APDU
INS	1 byte	Código de instrucción. Especifica la instrucción del comando
P1	1 byte	Parámetros de la instrucción. Proveen más información sobre la instrucción
P2	1 byte	
LC	1 byte	Número de bytes en el Data Field del APDU
Data Field	LC bytes	Secuencia de bytes con información
LE	1 byte	Cantidad máxima de bytes esperados como respuesta

Data Field	Hasta LE bytes	Secuencia de bytes con información
SW1	1 byte	Status Word (palabra de estado). Denotan el estado del procesamiento del comando en la tarjeta
SW2	1 byte	

Tabla 1.1

1.3.3. JavaCard Applet

Los applets son las aplicaciones que corren embarcadas en una JavaCard. Dichas aplicaciones interactúan en todo momento con el JCRE utilizando los servicios que éste brinda, e implementan la interfaz definida en la clase abstracta `javacard.framework.Applet`, la cual deben extender [SUN3].

Se puede decir que un applet comienza su ciclo de vida al ser correctamente cargado en la memoria de la tarjeta, linkeditada y preparada para su correcta ejecución [SUN1]. Una vez registrada en el JCRE (con el método `register()`, descripto más adelante), un applet está en condiciones de ejecutar. Este applet normalmente existe durante el resto de la vida de la tarjeta.[5]

La clase `javacard.framework.Applet` define cuatro métodos públicos que son utilizados por el JCRE para hacer funcionar las aplicaciones.

Método `install(byte[], short, byte)`

Este método es invocado por el JCRE antes de crear una instancia del applet en la tarjeta. La implementación usual de este método es llamar al constructor de la clase, que normalmente es privado, crear todos los objetos que el applet necesitará para su ejecución, y por último registrar el applet con el método `register()`. No es estrictamente necesario crear todos los objetos en el método `install()`. Sin embargo es una buena práctica de programación pues garantiza la obtención de toda la memoria necesaria, evitando quedar más adelante (tal vez una vez entregada al cliente) en un estado inválido por falta de memoria.

[5] Si bien algunas implementaciones permiten el borrado de applets, normalmente una vez que la tarjeta es distribuida, los applets que tuviese embarcados permanecen en la misma

En caso de que se produzca una excepción durante la ejecución del método `install()`, el JCRE es responsable de realizar las actividades de limpieza pertinentes. Una vez finalizado el método, el JCRE marca al applet como listo para ser seleccionado (ver método `select()`).

Método `select()`

Este método es invocado por el JCRE como consecuencia de la recepción a un SELECT APDU[6]. Este APDU, cuyo formato está definido en el ISO 7816, contiene el Application IDentifier (AID) del applet a seleccionar.

El AID es una secuencia de entre 5 y 16 bytes, que identifica de forma única una aplicación para SmartCards, de acuerdo al ISO 7816, y es la misma ISO la que otorga los AIDs. El formato de un AID se puede ver en la siguiente tabla:

Application Identifier (AID)	
National registered application provider RID	Proprietary application identifier extension PIX
5 bytes	Entre 0 y 11 bytes

Tabla 1.2

Cada empresa que produce applets debe solicitar a la ISO su propio RID, y a su vez maneja sus PIX (en forma arbitraria) para identificar sus aplicaciones y packages.

Una vez que el JCRE recibe un SELECT APDU, si hay algún applet seleccionado, invoca a su método `deselect()` (ver método `deselect()`) y luego invoca al método `select()` del applet cuyo AID fue especificado. El applet puede, por distintas razones, declinar la selección, en cuyo caso el JCRE es responsable de responder adecuadamente al CAD.

En caso de que la selección se realice sin inconvenientes, se pasa el SELECT APDU al método `process()` (ver método `process()`) del applet seleccionado para que lo procese y devuelva al CAD la información que sea pertinente.

Método `process(APDU)`

Cuando llega un APDU el JCRE invoca este método del applet seleccionado, pasándole como parámetro el COMMAND APDU recibido. Dentro de este método, el applet identifica el comando asociado al APDU y los parámetros, si los hay, y los procesa de acuerdo al protocolo que se haya definido para la interacción entre el applet y la aplicación terminal.

En caso de que la ejecución finalice correctamente, el applet sólo debe encargarse de cargar en el RESPONSE APDU la información que va a devolver, si la hay. El JCRE es responsable de setear los SW del RESPONSE APDU al valor especificado para ejecución exitosa (0x9000, de acuerdo a lo especificado en el ISO 7816).

Durante el proceso de un APDU, el applet puede levantar una `ISOException` con los SW apropiados, la cual, si no es atrapada por el código del applet, es atrapada por el JCRE, quien se encarga de generar el RESPONSE APDU correspondiente.

Método `deselect()`

Este método es invocado por el JCRE para avisar al applet que está actualmente seleccionado, que va a dejar de estarlo. Esto sucede cuando el JCRE recibe un SELECT APDU (aún cuando el AID del applet a seleccionar coincida con el del applet seleccionado).

[6] Que es un COMMAND APDU con cierta configuración (estándar) de CLA, INS y demás parámetros

Esto brinda al applet la oportunidad de realizar las tareas de limpieza que sean necesarias para quedar en un estado consistente.

1.3.4. Objetos Transitorios (Transient Objects)

En algunos casos, los applets utilizan cierta información que no necesariamente debe persistir entre dos sesiones diferentes, ya sea porque es temporal, o porque podría generar problemas de seguridad.

Es por esa razón que el JCRE permite definir objetos transitorios (transient objects), los cuales no mantienen su estado entre dos sesiones de CAD diferentes, o entre dos selecciones del applet al que pertenecen. Es importante notar que los objetos como tales siguen existiendo, y son persistentes; sin embargo, los valores de sus campos son seteados a sus valores por defecto (esto es, según la especificación de Java [SUN2, SUN7, SUN8], `0` para variables numéricas, `false` para variables booleanas y `null` para variables de referencia).

Los objetos transitorios de la plataforma JavaCard se dividen en dos categorías, a saber:

- *`CLEAR_ON_RESET` transient objects*: son aquellos que sólo son reseteados al producirse un reset del JCRE (por ejemplo, al insertar la tarjeta en el CAD).

- *`CLEAR_ON_DESELECT` transient objects*: son aquellos que son reseteados o bien al producirse un reset del JCRE o bien al ser recibido por éste un SELECT APDU estando el applet a la que pertenecen actualmente seleccionada.

La diferencia entre ambos es la capacidad de los `CLEAR_ON_RESET` transient objects de ser persistentes durante toda una sesión en el CAD, independientemente de si el applet se mantiene seleccionada durante toda la sesión o no. Esto puede ser útil a la hora de trabajar con claves de sesión o datos similares.

El JCRE debe implementar los objetos transitorios de forma que:

- Nunca queden almacenados en ningún medio permanente (como ser EEPROM)

- Sus campos deben ser reseteados a sus valores por defecto (según la categoría a la que pertenezca el objeto, se producirá el reseteo frente a diferentes eventos)

- El escribir o leer datos sobre los mismos no debe ser penalizado con un mayor tiempo de ejecución

- Las modificaciones en sus campos no son afectadas por transacciones. Esto es, nunca se recupera un dato que estaba contenido en un objeto transitorio, aún cuando se estuviera dentro de una transacción y la misma fuese abortada (ver transacciones).

1.3.5. Transacciones

El JCRE brinda la posibilidad de realizar transacciones, esto es, ejecutar un conjunto de sentencias en forma atómica, de forma que o bien se completa la ejecución, o bien se restaura el estado anterior de los objetos intervinientes.

El API brinda tres métodos para el manejo programático de transacciones:

- `JCSystem.beginTransaction()`: indica que comienza una transacción. A partir de la invocación a este método, el JCRE es responsable de mantener los valores anteriores de los campos modificados.

- `JCSystem.commitTransaction()`: indica que finaliza la transacción, y los cambios se hacen definitivos.

- `JCSystem.abortTransaction()`: indica que se finaliza anormalmente la transacción, y los estados originales de los objetos deben ser restaurados[7].

Adicionalmente, en caso de producirse un retorno (normal o anormal) de un método `select()`, `deselect()`, `process()` o `install()` mientras hay una transacción en progreso, se aborta la misma, y se restauran los estados originales de los objetos modificados. Un hecho muy importante a considerar es que, dado lo limitado de los recursos de la plataforma, existe una capacidad limitada de transacciones (*commit capacity*). El API provee métodos para determinar qué commit capacity hay disponible, aunque puede suceder que por problemas de overhead, la capacidad real sea menor que la indicada. En caso de que se exceda la capacidad máxima durante una transacción, se levanta una `TransactionException`.

1.4. Seguridad

1.4.1. Seguridad física

Prácticamente todo el mundo está de acuerdo en que las SmartCards son muy seguras y extremadamente difíciles de atacar, ya sea física o lógicamente. Algunos fabricantes [GEM, IBUT1] proclaman que sus tarjetas son físicamente inviolables, y que disponen de numerosas defensas contra ataques físicos, como ser detección de ciclos de reloj anormales en frecuencia, microprobing, retiro de la cubierta de resina epoxi o exposición del microprocesador a luz ultra violeta.

Sin embargo, hay claros ejemplos en los que se ha logrado obtener, a través de diversos medios, la información contenida en una SmartCard [AND1, AND2]. En la Universidad de Cambridge, un grupo de investigadores ha desarrollado varios métodos de extracción de información protegida dentro de una SmartCard, y los resultados que plantean no son muy alentadores. Según la clasificación de los posibles atacantes propuesta por IBM [AND1, AND2], un atacante de clase I, que podría ser un estudiante de grado de Ingeniería Eléctrica con menos de US$ 400 de equipamiento, podría lograr extraer información de una SmartCard.

De todos modos, los ataques registrados como exitosos sucedieron hace ya algún tiempo, y según los fabricantes, los componentes son cada vez más seguros.

1.4.2. Seguridad lógica

Las SmartCards multi-aplicación se están haciendo más y más populares. Los usuarios desean reducir el número de tarjetas en su billetera y las empresas emisoras desean reducir los costos de desarrollo y emisión de las mismas así como ampliar los servicios brindados. Además, el uso de SmartCards multi-aplicación permite la cooperación entre socios de negocios para lograr nuevas oportunidades de negocio.

La especificación de JavaCard provee ciertos mecanismos para la interacción entre applets residentes en la misma tarjeta, así como elementos que rigen la seguridad de dicha interacción, los cuales son presentados en la siguiente subsección.

1.5. Mecanismos de Seguridad Lógica

1.5.1. Applet Firewall

Es una tecnología que refuerza la seguridad más allá de las protecciones que tiene la JCVM por sí misma. Los chequeos que ésta implica se realizan durante la ejecución de la aplicación.

[7] Excepto los estados de los objetos transitorios, de acuerdo a lo explicado anteriormente

Se llamará *contexto* de un applet al conjunto de objetos que pertenecen a dicho applet. El JCRE también tiene su propio contexto, el cual tiene la misma estructura que el de un applet, teniendo además permisos especiales que le permiten realizar algunas operaciones a nivel de sistema que no son permitidas a los applets.

El applet firewall particiona el sistema de objetos de las JavaCards en los diferentes contextos de cada una de las applets que hay instalados en el dispositivo. El firewall se puede visualizar como la barrera que existe entre un contexto y los demás (Figura 1.4).

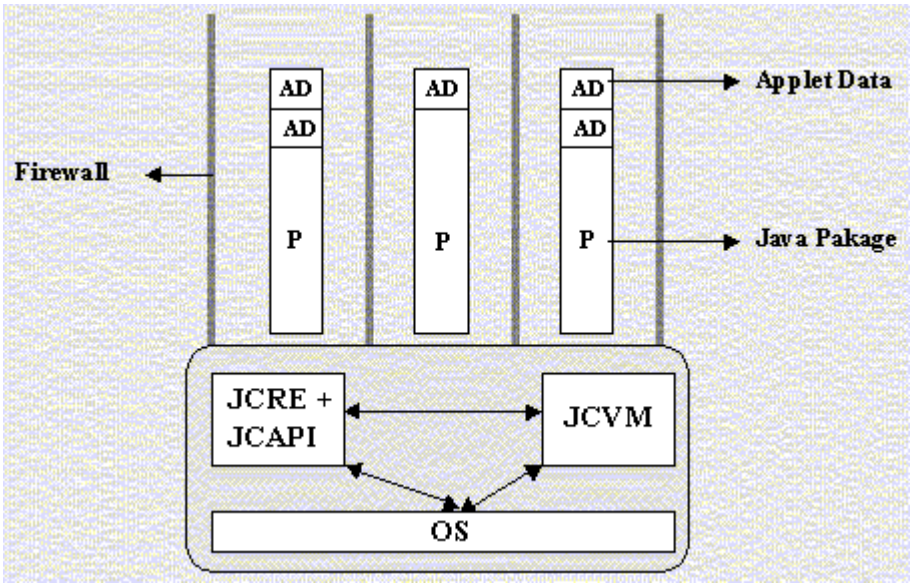


Figura 1.4

1.5.2. Context Switching

En todo momento sólo puede haber un contexto activo en la JCVM. Todo bytecode es chequeado en tiempo de ejecución para determinar si tiene permiso para acceder al contexto actualmente activo.

Bajo ciertas circunstancias, por ejemplo cuando un applet A invoca un método sobre un objeto compartido de otro applet B (ver Object Sharing), se produce un cambio de contexto (*context switch*). Cuando esto sucede, el contexto actual (contexto de A) se guarda en el stack de la JCVM y otro contexto (contexto de B) pasa a ser el contexto activo. Luego de que el método es ejecutado se restaura el contexto original guardando el contexto de B y restaurando el contexto de A para que quede como el contexto activo.

Los context switch pueden estar anidados, dependiendo la profundidad del anidamiento del espacio disponible en el stack de la JCVM.

1.5.3. Contextos Grupales

Este concepto fue introducido en la especificación 2.1 y permite que dos o más applets pertenecientes a un mismo package puedan compartir el mismo contexto. También todas las instancias de un mismo applet pueden compartir el mismo contexto.

De aquí en más se considerará que cada applet tiene su propio contexto.

1.5.4. Objetos

Cuando un objeto es creado, queda asociado al contexto activo. El objeto pertenece a la instancia del applet que lo creó o en caso que fuera un objeto creado por el JCRE éste es el dueño del mismo. Un objeto normalmente puede ser accedido sólo por el resto de los objetos que están en su mismo contexto. El firewall se encarga de controlar en tiempo de ejecución que los objetos no sean accedidos por applets (u otros objetos) que no estén en su mismo contexto.

Al intentar acceder a un objeto el contexto asociado al mismo se compara con el contexto activo para determinar si se tiene acceso a dicho objeto. En el caso de que no se tenga permiso sobre ese objeto se levanta una excepción (`SecurityException`).

Las siguientes subsecciones indican cómo puede un applet obtener la referencia a un objeto en otro contexto.

1.5.5. Protección del Firewall

El firewall intenta solucionar varios problemas de seguridad como:
- Errores del desarrollador o agujeros de seguridad en el diseño de los applets.
- Código incorrecto.

En caso de que un objeto en un contexto A obtenga una referencia a un objeto[8] en otro contexto B, no puede ejecutar ningún método del mismo, ni acceder a sus datos, ya que el firewall se lo impide.

1.5.6. Métodos o campos estáticos

Una clase no pertenece a ningún contexto, por lo cual en tiempo de ejecución no se realiza ningún chequeo en el caso de que se acceda a métodos o campos estáticos definidos en la misma. Estos pueden ser accedidos desde cualquier contexto. Los métodos estáticos se ejecutan en el contexto en el cual se originó la llamada a los mismos. Cabe destacar que los mecanismos de seguridad implementados por el firewall siguen vigentes para toda referencia a objetos que pueda ser obtenida de estos campos estáticos.

1.5.7. Acceso a objetos en otros contextos

El JCRE provee una serie de mecanismos para permitir la comunicación entre objetos que estén en diferentes contextos. Estos métodos son los siguientes:

JCRE Entry Point Objects (EPO): Esta es la facilidad que da el JCRE a los procesos del usuario para que realicen llamadas al sistema. El JCRE tiene una serie de objetos (en su contexto) que están marcados como Entry Points, lo cual habilita que los métodos de estos objetos puedan ser invocados desde cualquier contexto. Cuando esto sucede se realiza un context switch al contexto del JCRE. El contexto del JCRE es el único que puede tener este tipo de objetos. Hay dos tipos de Entry Point Objects:

Temporales. Los EPO de este tipo tienen todas las propiedades vistas hasta el momento, pero no se puede guardar referencias a los mismos en una variable de clase (o sea variables estáticas), variables de instancia o elementos de un array. Esto es controlado por el JCRE. Ejemplos de este tipo de EPO son los objetos APDU y las excepciones que pertenecen a la JCRE.

[8] Que no sea un objeto compartible; ver Object Sharing más adelante

Permanentes. La diferencia con los anteriores es que se puede guardar una referencia a los mismos. Ejemplos de este tipo de EPO son los objetos AID.

Global Arrays. Son simplemente arrays de objetos que son designados como globales por el JCRE (que es el único contexto que puede realizar esta designación), por lo cual pueden ser accedidos desde cualquier otro contexto. Estos objetos son temporales por lo que no se pueden guardar referencias a los mismos en variables de clase, variables de instancia o como componentes de un array (ya que estas referencias luego pueden pasar a un estado inconsistente). Los únicos arrays globales que hay hasta el momento en la especificación son los del buffer de APDUs y el buffer de parámetros del método `install()`.

1.5.8. Privilegios del JCRE

El JCRE tiene privilegios que lo habilitan a acceder a cualquier método o campo de cualquier objeto que esté en cualquier contexto. De todas formas cuando se ejecuta un método en otro contexto sigue realizándose el context switch.

1.5.9. Invocación de métodos sobre objetos en otros contextos

Otra forma de obtener acceso a objetos en otro contexto es utilizando la interfaz `Shareable` para definir *interfaces compartibles*. Este tipo de interfaces permite que un objeto que está en el contexto activo invoque métodos definidos en las mismas, e implementados por un objeto que se encuentra en otro contexto. Esta facilidad existía en una forma muy precaria en la especificación 2.0 [SUN5], pero fue rediseñada en la especificación 2.1. La siguiente subsección presenta este concepto en detalle.

1.6. Object Sharing

1.6.1. Introducción

El mecanismo de *Object Sharing* propuesto implica que para exportar cierto servicio a través del firewall se debe definir una interfaz que extienda la interfaz `Shareable` y luego implementar esta nueva interfaz en la clase que ofrecerá los servicios (Figura 1.5). A una instancia de esta clase se la llamará *Shareable Interface Object* (SIO). Los métodos de un SIO que estén declarados en una *Shareable Interface* (SI) pueden ser invocados por objetos que existan en otros contextos.

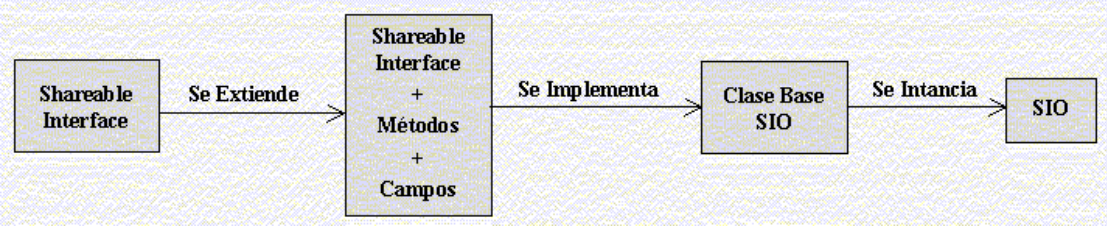


Figura 1.5

Cuando un objeto del contexto activo quiere hacer uso de algún servicio que brinde un SIO existente en otro contexto, debe invocar al método:

```
JCSystem.getAppletShareableInterfaceObject(AID, byte)
```

Esta invocación hace que el JCRE invoque un método del applet especificado por el parámetro AID, indicándole el AID del applet invocador. El método del applet es:

```
getShareableInterfaceObject(AID, byte)
```

El applet servidor puede devolver una referencia a un SIO o `null` en caso de que no acepte la solicitud. El objeto que pidió y obtuvo la referencia al SIO sólo puede hacer uso de los métodos y campos que son declarados en la SI que implementa la clase del SIO. El firewall no le permite acceder al resto de los métodos que pueden estar definidos en el SIO.

1.6.2. Detalle del funcionamiento

Los pasos que se deben seguir para lograr la comunicación entre dos applets en distintos contextos utilizando Shareable Interfaces son los siguientes. Se llamará *servidor* al applet que brinda un servicio y *cliente* al applet que lo solicita.

Servidor	Define una SI, extendiendo la interfaz `javacard.framework.Shareable`, y declarando en esta extensión los métodos y propiedades que se quieren ofrecer a los clientes.
Servidor	Define una clase que implemente la interfaz creada en el paso anterior.
Servidor	Crea una instancia de la clase definida en el paso anterior.
Cliente	Declara una variable para guardar una referencia a un SIO del tipo que quiere obtener (normalmente el tipo definido por la extensión de la SI).
Cliente	Invoca el método: `JCSystem.getAppletShareableInterfaceObject()` para pedir un SIO del Applet Servidor (como se ve en el paso 1 de la Figura 1.6). A este método le pasa el AID del applet servidor del cual quiere el SIO y que interfaz es la que quiere.
JCRE	Invoca el método `getShareableInterfaceObject()` del applet servidor pasándole el AID del cliente, así como la interfaz que pidió. Este método se ejecuta en el contexto del servidor.
Servidor	En base al AID que recibe decide si comparte o no el SIO. En caso de que decida compartirlo, retorna una referencia al SIO. En caso contrario devuelve `null`.
JCRE	Pasa el resultado al cliente invocando al método: `GetShareableInterfaceObject()`
Cliente	Guarda la referencia recibida[9]

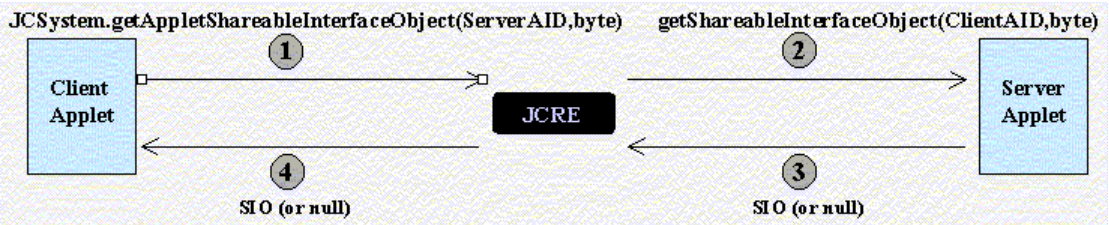


Figura 1.6

Luego que el cliente tiene la referencia del SIO puede invocar los métodos que están definidos en la interfaz que éste implementa. Cuando se realiza una invocación se produce un context switch al contexto del SIO.

[9] Si bien la especificación del JCRE no especifica si las referencias a SIOs pueden ser persistentes o no, desde un punto de vista de seguridad es mejor que no lo sean, y que sean solicitadas en cada sesión.

El SIO puede determinar el AID del cliente que requiere el servicio invocando al método `JCSystem.getPreviousContextAID()`.

Al haber un cambio de contexto el método invocado tiene acceso a todos los objetos dentro de su propio contexto. Esto permite que un cliente acceda a ciertos objetos privados, pero sólo a través de una interfaz bien definida, y con código implementado en el servidor.

Luego de que se termina de ejecutar el método invocado se restaura el contexto anterior.

1.6.3. Consideraciones acerca de object sharing

El mecanismo de object sharing descripto en la subsección anterior presenta algunas dificultades, que son discutidas más profundamente en [MONKRI], Sección 3 y Apéndice I.

Sección 2
Herramientas de Desarrollo

2.1. Emulador

Como parte de la investigación realizada sobre la tecnología JavaCard, y antes de contar con hardware adecuado, se experimentó con el kit de desarrollo brindado por Sun [SUN4]. El mismo consiste de varios componentes (básicamente clases, verificador de bytecode, conversor y emulador), los cuales permitieron realizar los primeros experimentos de desarrollo de applets. Las conclusiones sobre el trabajo con el kit de Sun son presentadas en la sección 3y en el Apéndice I. A partir de ciertas dificultades experimentadas con el mismo, se decidió implementar el paquete de generación de código descripto a continuación.

Para la utilización del emulador de JavaCard provisto por Sun se requiere de dos componentes, a saber:

- el emulador propiamente dicho
- una herramienta de scripting de APDUs (apdutool)

El emulador se ejecuta y queda a la espera de APDUs en un puerto dado. El apdutool se conecta a dicho puerto, y actúa como driver del CAD, enviando APDUs que toma de un script y generando un log con las respuestas del emulador. El script consiste básicamente de una secuencia de strings que representan números hexadecimales.

2.2. Proxy

2.2.1. Motivación

La utilización del emulador de Sun presenta algunos problemas:

- No permite interacción con el applet, ya que el script representa una sesión completa entre el CAD y la tarjeta.
- La generación de los APDUs es manual, ya que no hay herramientas adecuadas en el kit para esta tarea[10]. Esto resulta extremadamente tedioso y muy propenso a errores.

Para solucionar estos problemas se decidió implementar una herramienta que sustituyera al apdutool, interactuando directamente con el emulador desde una interfaz que pudiera ser manipulada interactivamente por el usuario.

La implementación de esta solución presentaba nuevos problemas, a saber:

- Debido a la falta de documentación referente al protocolo utilizado entre el apdutool y el emulador, sería necesario utilizar algún mecanismo de ingeniería reversa (en este caso probablemente la intercepción y el estudio de las comunicaciones entre el apdutool y el emulador) para poder entender completamente el funcionamiento del protocolo, y luego poder desarrollar el driver necesario.
- Una interfaz de usuario permitiría lograr interactividad en el testeo de las applets, pero la creación de los APDUs seguiría siendo manual, o reprogramada de alguna forma para cada applet.

[10] Sí las hay provistas por terceros [TL], pero no se dispuso de las mismas durante el transcurso de este proyecto.

- La programación de la generación de APDUs sería prácticamente tan tediosa y propensa a errores como el ingreso manual de los mismos.

Se distingue dos partes importantes de los problemas encontrados, éstas son:

- Encapsulación de la generación de APDUs en una interfaz de mayor nivel
- Generación de drivers para el emulador, y eventualmente otros dispositivos

La primer parte fue atacada en primera instancia, y dio lugar a una herramienta llamada *Proxy*. El segundo problema no fue atacado directamente, proponiéndose como trabajo futuro. La razón principal de esta omisión es la obtención del hardware adecuado.

2.2.2. Diseño

El primer problema atacado fue el de la encapsulación del protocolo de comunicación entre las aplicaciones de terminal y los applets. Para ello se utilizó el concepto de invocación de procedimientos remotos (similar al RPC de Sun, o CORBA). La completa abstracción de la comunicación entre applet y terminales tiene como ventaja que permite aumentar la velocidad de desarrollo tanto de las aplicaciones de terminal, como de los applets, disminuyendo además la cantidad de errores debido al manejo directo de APDUs.

Este concepto ya fue implementado por Gemplus en su kit GemXpresso 211 [GX211]. Sin embargo la solución requiere modificaciones del JCRE y el ambiente de desarrollo es propietario. Si bien las tarjetas de Gemplus soportan el manejo estándar de APDUs, las extensiones propietarias introducidas no son soportadas por otros fabricantes, lo cual implica problemas de compatibilidad serios.

Un análisis del funcionamiento de un applet, permite ver fácilmente que el mismo se basa en la codificación y selección de los métodos que el applet exporta a la aplicación de terminal, utilizando códigos de instrucción, y haciendo un proceso de marshaling de los parámetros y los valores de retorno.

Normalmente, para la implementación de mecanismos de invocación remota se utiliza un lenguaje de definición de interfaces (IDL) independiente del lenguaje en que se implementen los procedimientos exportados. En este caso se decidió utilizar el propio código Java del applet, extendido con comentarios especiales, como IDL. El applet es procesado posteriormente por el generador de código. El mismo produce un objeto (*proxy*) que representa al applet en la aplicación terminal, y la implementa del método `process(APDU)` del applet junto a algunos métodos auxiliares para el marshaling / unmarshaling de parámetros y resultados.

De esta forma, cuando un método del proxy es invocado por la aplicación en el terminal, genera una *invocación*, que es pasada por el proxy a una *conexión*, que implementa el driver para el dispositivo que se vaya a utilizar (CAD o emulador). La conexión realiza el marshaling y transmite el APDU a la tarjeta. Esto resulta en una invocación al método `process()` del applet. Este se encarga de invocar al método adecuado en el applet y de generar el resultado. Este resultado sigue el camino inverso, volviendo al proxy como un objeto *resultado*, a partir del cual se devuelve un valor al invocador del método.

Junto con el generador de código, se provee un conjunto de clases adicionales que permiten abstraer la tarjeta y el CAD en sí, como si fueran objetos locales a la aplicación de terminal. El diseño permite la fácil sustitución de algunos de los componentes lo que permite extenderlo para su uso a distintos tipos de terminal. Si bien el proxy generado y el resto de las clases están implementadas en Java, es sencilla su implementación como JavaBeans. Esto permitiría utilizarlos como componentes desde otros lenguajes (como por ejemplo, Visual Basic, Visual C++, C++, etc.) si esto fuera necesario.

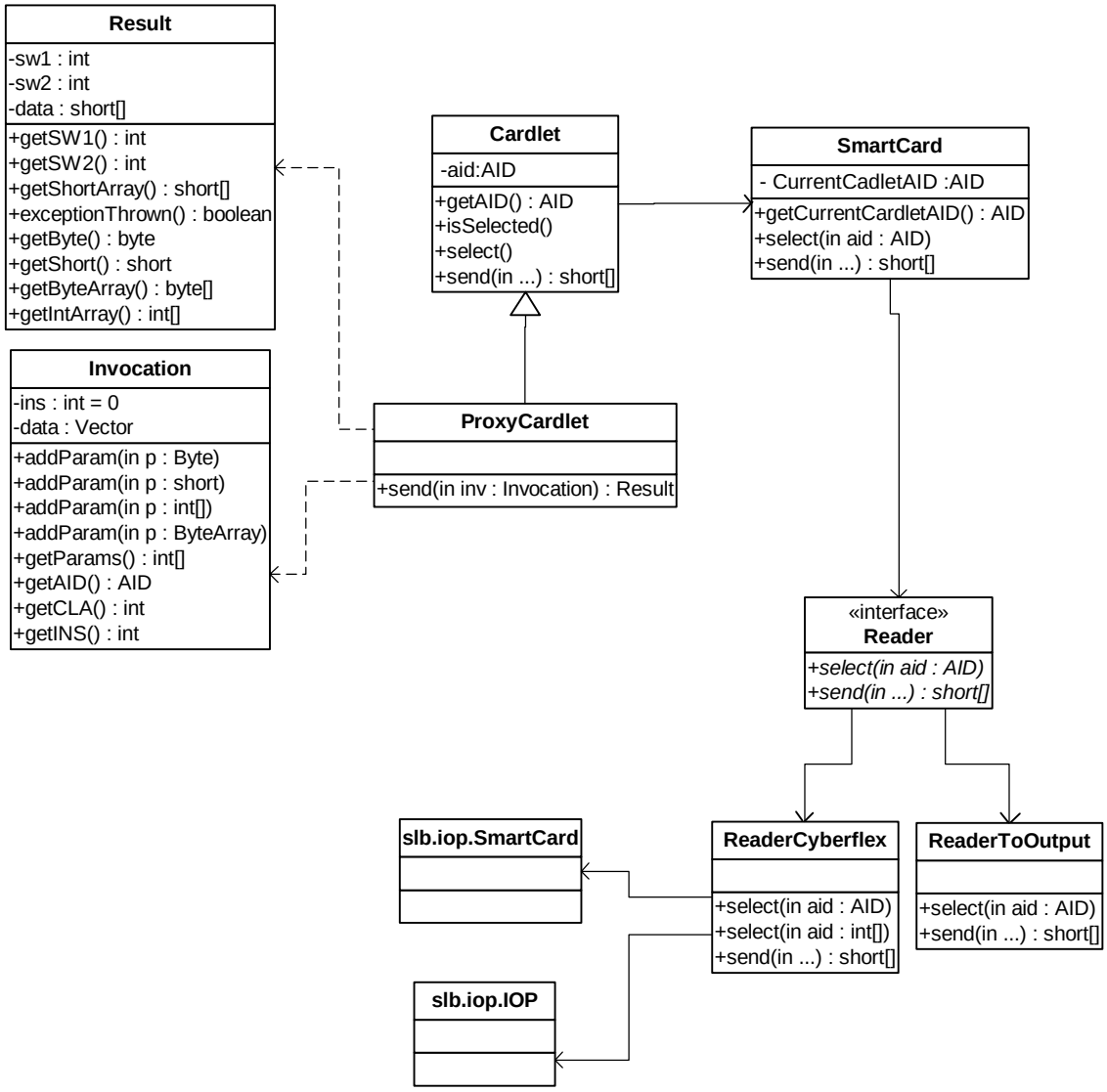


Figura 2.1

Mediante la utilización de esta herramienta, el desarrollador del applet puede programar la misma como si fuera un objeto cualquiera (respetando las limitaciones de la JCVM, por supuesto), y el desarrollador de la aplicación de terminal trabaja sobre el applet como si fuera un objeto local a la terminal. Esto contribuye a la disminución (probablemente importante) del tiempo y la dificultad del desarrollo.

2.2.3. Implementación

El prototipo implementado es básicamente una prueba de concepto, pero brinda los siguientes servicios:

- Generación del proxy y el método `process()` para el applet
- Generación del método `process()` del applet, junto con los métodos auxiliares necesarios (incluyendo sólo los que son necesarios, de forma de minimizar el footprint del applet)
- Manejo de bytes, shorts y arrays de bytes
- Manejo de excepciones, permitiendo que el usuario defina las excepciones que pueden lanzar sus métodos.

El prototipo original fue escrito con base en la especificación 2.1 del JCRE, pero dado que el hardware del que se dispone se basa en la especificación 2.0, y tiene algunas características propietarias, se reescribió para adaptarlo a la misma. La mayor parte de las modificaciones necesarias fueron debido a cambios en el API desde la versión 2.0 a la 2.1.

2.2.4. Resultados obtenidos

Los resultados obtenidos hasta la fecha son parciales pero alentadores. Si bien el código generado todavía presenta algunos problemas que impiden su correcto funcionamiento, se está trabajando para solucionarlos. El desarrollo del proxy ha contribuido a un mejor entendimiento del kit que está siendo utilizado, el cual es descripto con más detalle en la subsección 2.3.

En el Apéndice II se presenta un pequeño caso de estudio en el cual se utilizó el proxy, para brindar más detalle sobre su funcionamiento.

2.2.5. Trabajo futuro

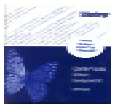
La implementación actual tiene varias limitaciones, y puede realizarse numerosas mejoras. Entre las mejoras a introducir se destaca el manejo de strings (que si bien no son soportados por la JCVM, se pueden codificar trivialmente como arrays de bytes) y eventualmente de otros tipos de datos más complejos, implementar algún otro back-end para el generador de código (posiblemente para Visual Basic, para el cual el kit de Schlumberger con el que se cuenta tiene soporte) y estudiar la posibilidad de un mejor manejo de excepciones en el applet. Debido a ciertos detalles del API de JavaCard y a una simplificación en el diseño del prototipo, actualmente todos los métodos exportados deben declarar que lanzan cierto tipo de excepción, aunque no lo hagan.

2.3. Kit

2.3.1 Reader

El reader que se utilizó es un producto de Schlumberger llamado Reflex 72 RS232 Reader. Utiliza el puerto serial para transferencia de datos y un puerto PS/2 para la energía, pudiéndose conectar el mouse o el teclado en el mismo puerto PS/2. Se comunica con la SmartCard a través de firmware embebido en el reader. Es compatible con PC/SC (ver sección 2.3.4) por lo que cualquier aplicación compatible con PC/SC puede utilizar este reader sin modificaciones en la aplicación.

2.3.2. Kit de Desarrollo



Para el desarrollo se utilizó un kit de desarrollo de Schlumberger [SLB2], que incluye el reader descripto anteriormente, dos Cyberflex SmartCards y dos Criptoflex SmartCards.

El kit, llamado Cyberflex Access SDK, provee las siguientes facilidades:

- Interacción con la tarjeta a nivel de APDU
- Posibilidad de realizar scripts de APDU
- Monitoreo de la comunicación entre la tarjeta y el reader
- Framework para el desarrollo de aplicaciones que necesitan comunicarse con las tarjetas Cyberflex y Criptoflex

2.3.3. Cyberflex y Criptoflex SmartCards

Cyberflex es una SmartCard, de 16 KB de EEPROM, que permite utilizar la programación en Java para el desarrollo de las aplicaciones a embarcar. Contiene un microprocesador que provee capacidad de procesamiento y memoria para almacenar instrucciones y datos.

Es una Java SmartCard (JavaCard). Contiene una máquina virtual llamada Solo y un sistema operativo. La máquina virtual Solo (SVM) es un intérprete que traduce bytecode de Solo en instrucciones que pueden ser entendidos por el sistema operativo en la tarjeta. La SVM lleva a la SmartCard la flexibilidad y portabilidad provista por el lenguaje de programación Java.

La implementación actual de SVM soporta la especificación JavaCard API 2.0 de Sun, y brinda soporte para múltiples aplicaciones. Sin embargo no soporta mecanismos de object sharing ya que estos fueron introducidos en la especificación JavaCard API 2.1 de Sun.

Sus características de seguridad incluyen comandos externos e internos para la autenticación utilizando DES, Triple DES o RSA. Permite utilizar los servicios criptográficos que brinda a nivel del API de Java embarcado en la tarjeta. Los programas así desarrollados pueden ser firmados digitalmente para ser embarcados en la tarjeta, la cual controlará su firma antes de proceder a la instalación del programa.

El kit de desarrollo mencionado antes brinda una herramienta llamada MakeSolo que verifica y traduce bytecode de Java (archivo .class) a bytecode de SVM (archivo .bin). Este último es el archivo que se embarca en la Cyberflex SmartCard.

2.3.4. Compatibilidad PC/SC

PC/SC es un workgroup cuyo objetivo es el de promover una especificación estándar, que asegure la interoperabilidad entre SmartCards, SmartCard readers y computadoras.

PC/SC desarrolló una especificación independiente de la plataforma, que puede ser implementada sobre cualquier sistema operativo. Fue construido sobre los estándares actuales de SmartCard (ISO 7816), definiendo interfaces de bajo nivel para dispositivos y APIs independientes del dispositivo.

La especificación actual es la PC/SC Specification 1.0. La misma incluye los siguientes temas, entre otros:

- Provee la arquitectura del sistema y de sus componentes
- Detalla las características y requerimientos de compatibilidad de las tarjetas y los dispositivos
- Presenta una discusión con consideraciones de diseño para los dispositivos, y recomendaciones de implementación
- Describe los componentes con los que debe contar el sistema
- Presenta consideraciones de diseño para desarrolladores de aplicaciones, indicando como hacer uso de los componentes

2.3.5. Testeos

La primer aplicación de testeo desarrollada es una aplicación de simulación de un teléfono público. Soporta tres comandos, uno para *decrementar*, que decrementa la cantidad de cómputos en 1. Otro para *cargar*, que permite agregar cómputos a la tarjeta. Y por último un comando para *consultar* la cantidad de cómputos en la tarjeta.
Se desarrolló el applet en Java, y el mismo consta de unas 100 líneas de código. La aplicación de terminal fue desarrollada en Visual Basic. La misma detecta la inserción de una tarjeta, consulta la cantidad de cómputos, decrementa periódicamente los cómputos, y permite recargar la tarjeta.

Se desarrolló otro applet más complejo cuyo objetivo es investigar el protocolo de comunicación entre un CAD y una SmartCard tradicional. El applet recibe un APDU, lo guarda en memoria, y devuelve un código de error. Luego se reimplementa el applet para que responda adecuadamente a dicho comando y guarde el próximo. Para averiguar qué debe responder la tarjeta se puede utilizar el reader con un programa que mande el APDU adecuado a la tarjeta (a una original), y así detectar su respuesta. Mediante este mecanismo es posible averiguar el protocolo completo.

2.3.6. Problemas detectados

2.3.6.1. Kit

Para el desarrollo de la segunda aplicación de testeo se requería que el applet estuviera seleccionada por defecto en la tarjeta, debido a que la aplicación de terminal asumía que era una tarjeta que estaba dedicada a un único applet. Para ello se utilizó una facilidad de las tarjetas CyberFlex que permite preseleccionar un applet para que se seleccione por defecto al resetearse la tarjeta.

Sin embargo, el kit de desarrollo utilizado presenta un problema cuando la tarjeta tiene un applet seleccionado automáticamente, y hace que no se pueda trabajar normalmente sobre la tarjeta. Este problema fue reportado previamente y la solución está publicada en un foro de discusión de Schlumberger [FORO].

2.3.6.2. Cyberflex SmartCards

Se detectaron varios problemas en el trabajo con las tarjetas Cyberflex.

Una implementación usual de un applet declara un juego de constantes para representar el CLA y cada uno de los INS a utilizar. Luego, en el método `process()`, se chequea que el CLA sea el correspondiente y se realiza una selección según el INS recibido. Ciertos valores de CLA e INS presentan problemas al momento de ejecutar, como por ejemplo, no seguir la rama del switch adecuada al valor enviado. En un principio el error fue complejo de detectar, pero su solución es simple, encontrar un juego de constantes que no tenga problemas.

Otro problema detectado es la selección de una celda de un arreglo. No puede seleccionarse un arreglo utilizando `byte`, sino que debe ser `short`.

Por último, en ciertas ocasiones el hecho de realizar una invocación a `throwit()` en el código del applet no retorna con los valores SW1 y SW2 adecuados. El JCRE brinda un método llamado `throwit()` que se utiliza para los casos en que ocurre una excepción y el CAD debe ser notificado. El Response APDU cuenta con dos bytes (status word) que indica el código de error, o 0x90 0x00 en caso de ejecución exitosa. El método `throwit()` recibe como parámetros dos bytes, correspondientes al status word a retornar. El problema que se detectó es que en ciertas ocasiones el status word recibido no es el esperado.

Sección 3
Una metodología para el uso seguro de objetos compartidos

3.1. Presentación del problema

Uno de los aspectos claves a tener en cuenta en el desarrollo de aplicaciones JavaCard es la seguridad de los datos que las mismas manejan, debido a que los mismos son potencialmente muy sensibles (datos médicos, financieros, control de acceso a sistemas restringidos, etc.).

Debido a ello, la especificación del JCRE provee mecanismos de aislamiento de las aplicaciones que coexisten en un misma tarjeta [SUN1]. Esto permite a los desarrolladores programar con la tranquilidad de que sus datos están a salvo dentro de la tarjeta.

Sin embargo, hay ciertas aplicaciones que se pueden beneficiar de la interacción entre applets. Ejemplos de la utilización de la interacción entre applets son los sistemas de lealtad, donde varios comercios asociados trabajan con el mismo mecanismo de puntos, o en los casos en que varias applets utilizan un mismo servicio de autenticación del usuario, que puede estar implementado como un único objeto para ahorrar espacio.

Es por ello que el JCRE provee ciertos mecanismos de interacción entre applets, para permitir que las mismas compartan ciertos datos o servicios, definidos por el desarrollador. De esta forma se logra una mayor flexibilidad en el desarrollo de applets, y se abre el campo a nuevas aplicaciones basadas en las interacciones inter-applet.

La especificación 2.1.1 del JCRE [SUN6] (la más actual a la fecha) define (al igual que la 2.1) interfaces para permitir que las applets exporten ciertos objetos a través de las protecciones que impone el JCRE. Dichas interfaces presentan ciertas limitaciones y problemas de seguridad, que son señaladas en [MONKRI].

El mecanismo de object sharing especificado en el JCRE 2.1.1 se basa en la definición de *shareable interfaces* (SIs). Se extiende la interfaz `javacard.framework.Shareable`, definiendo así nuevas shareable interface, y se desarrolla las clases que las implementa. Las instancias de estas clases se denominan *Shareable Interface Objects* (SIOs). Estos objetos son los que permiten la interacción entre applets, ya que los métodos declarados en una SI pueden ser ejecutados a través del firewall, en el contexto del servidor, logrando de esta forma acceso a los datos y servicios que éste brinda. En el caso de una aplicación de lealtad, por ejemplo, la SI podría definir métodos para incrementar, decrementar y consultar el número de puntos acumulados por un cliente, así como verificar la identidad del mismo.

Los applets que actúan como servidor implementan un método que permite exportar los SIOs necesarios para brindar servicios a otros applets (clientes). El método a implementar es `getShareableInterfaceObject()`.

Dichos clientes acceden al SIO que requieren, a través de un método del JCRE que recibe como parámetros el AID del applet servidor, y un byte para especificar opciones. El método a invocar es `JCSystem.getAppletShareableInterfaceObject()`. Este método invoca al método `getShareableInterfaceObject()` del applet indicado, que devuelve una referencia al SIO solicitado, o `null`, en base al AID del cliente, el cual recibe como parámetro.

Con la referencia al SIO, el cliente puede invocar métodos sobre el mismo, sin que el firewall se interponga, y de esta forma obtener los servicios o datos necesarios del servidor.

Este mecanismo presenta los siguientes problemas:

- La autenticación del cliente se basa en su AID, lo cual implica que:
  - La cantidad (e identidad) de clientes que atenderá un servidor en su vida útil está determinada al momento de embarcarlo en la tarjeta
  - Si la tarjeta fuese comprometida, se podría utilizar un applet malicioso instalado con el AID del cliente para extraer datos del servidor.

- Es común la implementación de varias interfaces en un mismo objeto (normalmente el propio applet), lo que permite a un cliente malicioso realizar un cast de una interfaz a otra, obteniendo así acceso ilícito a datos o servicios de la misma.

- Los métodos del SIO no pueden recibir objetos (que no sean a su vez SIOs como parámetros, ya que el firewall no permite la ejecución de ningún método sobre los mismos)

La solución planteada en [MONKRI] resuelve los dos primeros problemas, dejando el tercero de lado. En realidad el tercer problema no está relacionado con la seguridad de las aplicaciones, sino que es una limitación a la hora de desarrollarlas.

El trabajo descripto en esta sección dio lugar a la generación de un artículo de divulgación, que ha sido publicado como reporte técnico del PEDECIBA – In.Co. [PRV], y que será sometido a una conferencia internacional a la brevedad. Para obtener una descripción más detallada de la metodología propuesta referirse al Apéndice I de este documento, en el cual se incluye dicho artículo.

3.2. Una posible solución

En [MONKRI] se propone una serie de modificaciones a la especificación, que tienen como objetivo corregir los defectos de la versión actual. La idea central es sustituir el mecanismo de object sharing actual por un sistema de *objetos delegados*, en el cual cada applet servidor tiene un delegado que exporta sus servicios. Dicho delegado es accesible desde otros contextos, y es a través de él que los clientes acceden a los servicios del servidor. A su vez, es el delegado de cada cliente el que se encarga de comunicarse con el delegado del servidor, de forma que sólo hay interacciones entre objetos delegados.

El delegado del servidor sólo permite el acceso a los métodos que se desean exportar, y establece los mecanismos de autenticación necesarios para cada método, de forma de poder determinar, mediante un mecanismo de challenge-response con un secreto compartido, si el cliente que solicita acceso a un método está autorizado a obtenerlo. De esta forma, se logra solucionar las fallas en el mecanismo de seguridad de la implementación actual:

- El problema del applet malicioso instalado con un AID válido desaparece, ya que el mecanismo de autenticación no se basa en el AID, sino en un esquema de challenge-response
- Por el mismo motivo desaparece la limitación en el número de posibles clientes
- Finalmente, no hay posibilidad de realizar un cast entre interfaces compartibles, dado que existe una única interfaz exportada (la que brinda el objeto delegado), y el cliente es autenticado en cada invocación

Las modificaciones propuestas, si bien solucionan los problemas presentes en la actual versión del JCRE, implican cambios muy significativos en el mismo, y eventualmente pueden producir serios problemas de compatibilidad con hardware y aplicaciones basados en la especificación actual.

3.3. Un enfoque metodológico

Como parte de este trabajo, y con una fuerte base en [MONKRI] se planteó una metodología de desarrollo, que utilizando las interfaces provistas por el JCRE, permite solucionar los mismos problemas que soluciona el enfoque de objetos delegados, sin necesidad de realizar modificaciones en la especificación, lo que es claramente deseable.

El concepto central en el enfoque de los objetos delegados es el esquema de autenticación de los clientes. Si bien los posibles ataques sobre la plataforma actual son extremadamente difíciles de realizar si existe una política de seguridad adecuada en el proceso de desarrollo y distribución de los applets [GIR], la autenticación por medio del AID del cliente es débil y limitante. Se puede, sin embargo, implementar un esquema más seguro utilizando interfaces compartibles y SIOs, sin necesidad de modificar la especificación.

La metodología desarrollada establece que para obtener acceso a los distintos servicios brindados por un applet servidor, el cliente debe autenticarse previamente. Para ello el servidor ofrece un SIO, llamado SecureSIO (SSIO), que brinda los métodos necesarios para implementar un mecanismo de challenge-response (definidos en una SI llamada *SecureSI*), similar al utilizado por los objetos delegados. Este objeto es accesible por todos los potenciales clientes, que una vez autenticados quedan registrados en el servidor durante el resto de la sesión, pudiendo acceder de esta forma a los servicios requeridos. Es posible, y de hecho así se hizo en el caso de estudio, la inclusión en este esquema de un *Authorization Manager* (AMgr), que lleve un registro de todos los clientes de la sesión, y de los privilegios de los mismos (a qué SIOs y eventualmente a qué métodos pueden acceder, si es necesario tal nivel de granularidad). De esta forma el servidor puede realizar un adecuado control de acceso a sus servicios.

La utilización de un SSIO permite solucionar el problema del applet malicioso con un AID válido, y el de la limitación de atención sólo a clientes conocidos al embarcar el servidor, mediante el mecanismo de challenge-response para la autenticación del cliente.

El problema del cast inapropiado de una interfaz a otra se puede resolver en forma sencilla, implementando cada interfaz exportada en una clase distinta, y de forma que estas clases no estén en la relación de subclase. De esta forma cada SIO sólo permite el acceso a los métodos de la interfaz que implementa. Cabe notar que esto implica una mayor cantidad de código para almacenar, lo cual es una desventaja. De todas formas esto no es un problema mayor si se implementa bien.

La metodología propuesta fue utilizada para la implementación de un caso de estudio trivial como prueba de concepto. Por no contar con hardware que se ajuste a la especificación 2.1.1, el caso de estudio fue testeado sobre software de simulación [SUN4], que de por sí es limitado en funcionalidades [SUN4]. Por esta razón los resultados obtenidos hasta la fecha, si bien son alentadores, no son concluyentes.

En la siguiente sección se presentan algunas conclusiones extraídas del trabajo realizado.

3.4. Conclusiones

Si bien la experimentación realizada no fue extensa, se obtuvieron algunas conclusiones interesantes, tanto con respecto a la propuesta metodológica planteada, como con respecto al kit de desarrollo de Sun.

Con respecto al mecanismo de object sharing existente en la especificación actual, se puede concluir que el mismo presenta serias fallas en los que respecta a la seguridad, e incluso a nivel de funcionalidad, dada la imposibilidad[11] de pasar referencias como parámetros en invocaciones a métodos de otro applet.

[11] Sin implementar como SIOs todos los objetos que pudiera interesar pasar como parámetro.

También se observó que existen al menos dos soluciones para resolver los problemas de seguridad existentes: una que implica modificaciones en la especificación y otra que toma un enfoque metodológico, obteniendo ambas resultados similares. Esto implica que en principio no es necesaria una modificación a la especificación del JCRE para solucionar los problemas de seguridad mencionados, sino que basta con adoptar una determinada metodología de desarrollo.

Se notó también que los ataques sobre applets que utilicen el mecanismo de object sharing, si bien son viables, se ven enormemente dificultados si existe una adecuada política de seguridad en torno al desarrollo y deployment de las aplicaciones.

En lo relativo al problema del pasaje de referencias como parámetros en invocaciones sobre SIOs, no se logró encontrar una solución metodológica. Considerando que el mencionado problema requeriría un estudio mucho más profundo para determinar la factibilidad de una solución metodológica, y que no afecta directamente a la seguridad, se decidió no atacarlo en el marco del presente trabajo.

Finalmente, se obtuvo cierta experiencia con el kit de desarrollo brindado por Sun. De acuerdo a los trabajos realizados sobre el mismo, se puede concluir que es bastante limitado, y podría ser mejorado notoriamente mediante la inclusión de mejores herramientas de debugging y scripting, y permitiendo la emulación de todas las funcionalidades de la plataforma.

Sección 4
Caso de Estudio

4.1. Objetivos

El caso de estudio se centra en desarrollar un prototipo de un sistema para el manejo de historias clínicas resumidas, utilizando como medio de almacenamiento una SmartCard. En particular la implementación va a ser desarrollada para una JavaCard, de forma de poner en práctica los conocimientos obtenidos y evaluar las herramientas de desarrollo.

Los tres objetivos fundamentales son:

- Definir el contenido básico de una historia clínica resumida. Actualmente a nivel nacional no existe ninguna propuesta de especificación de este tipo de documento.
- Desarrollar un prototipo funcional de una historia clínica resumida utilizando la tecnología JavaCard. Consta de implementar un applet que brinde las funcionalidades necesarias para manipular la información de una historia clínica resumida.
- Desarrollar una aplicación para estaciones de trabajo que permita interactuar con la JavaCard, permitiendo así manipular la información almacenada en ella.

4.2. Problema - Análisis de requerimientos

4.2.1. Descripción general

El caso de estudio fue enmarcado en un proyecto general del Instituto de Computación con el Hospital de Clínicas. Varios proyectos en este marco tienen como objetivo la informatización de la historia clínica de los pacientes del Hospital de Clínicas.

El enfoque propuesto por nuestro trabajo fue diferente respecto de los proyectos en dicho marco, basándose en lo que se conoce como Health Card. Una Health Card es una SmartCard que contiene información personal, médica y de seguridad utilizada por los sistemas médicos. La tarjeta puede ser leída por SmartCard readers ubicados en los sistemas partícipes, por ejemplo: hospitales, sistemas de emergencia médica móvil, farmacias, clínicas, consultorios particulares, etc. Las personas que participen del sistema llevan consigo su tarjeta, la cual guarda información importante, accesible en forma inmediata. La información médica puede incluir medicación actual, alergias, tratamientos, etc., pudiendo incluir además información personal del paciente. Esta tecnología está en uso en varios países (un ejemplo de estos sistemas puede encontrarse en [INET1, INET2]).

4.2.2. Análisis de requerimientos

4.2.2.1. Fuentes

El análisis de requerimientos del caso de estudio tuvo como fuente principal varias reuniones con el Dr. Luis Martínez, del Hospital de Clínicas, asistencia a eventos de informatización de historias clínicas (SUIS - Sociedad Uruguaya de Informática en la

Salud), documentación publicada por el Dr. Roberto Rocha, que está siendo utilizada en Brasil [HCR1, HCR2], e información en Internet.

En otros países se utiliza la tecnología de SmartCard aplicada a historias clínicas con un fin diferente al propuesto por nosotros. Su principal uso es por concepto de reducción de gastos en papelería y trámites que una persona realiza cada vez que se atiende, así como para evitar fraudes [INET3].

El enfoque que se ha tomado en este trabajo difiere con el arriba descripto: se detectó que la utilidad del sistema para el propio Hospital de Clínicas era baja, ya que una red local con un sistema de información adecuado satisface de mejor manera las necesidades planteadas. La SmartCard brinda un valor agregado cuando no es posible tener la información centralizada; por ejemplo en sistemas de emergencia móvil y en sistemas de salud descentralizados con pequeños consultorios remotos sin conexión a un sistema central.

4.2.2.2. Requerimientos

Un primer análisis detectó que el principal objetivo era determinar qué se entiende por historia clínica, en particular por historia clínica resumida.

Dado la escasez de recursos de memoria en la tarjeta fue de gran interés definir con certeza los datos a ser almacenados, formas de codificación de los mismos y estructuras de datos que permitiesen su almacenamiento en forma eficiente.

La información de interés sobre el paciente está comprendida en las siguientes categorías:

- Datos administrativos
- Datos de la institución de salud a la que está afiliado el paciente
- Alergias y reacciones adversas
- Enfermedades crónicas
- Diagnósticos realizados
- Procedimientos realizados
- Fármacos en uso

La sección (iii.2) del Apéndice III muestra en detalle la información dentro de cada categoría y presenta un análisis del tamaño de la misma.

El siguiente diagrama de estructura estática de UML presenta la jerarquía de clases que modela la realidad del problema:

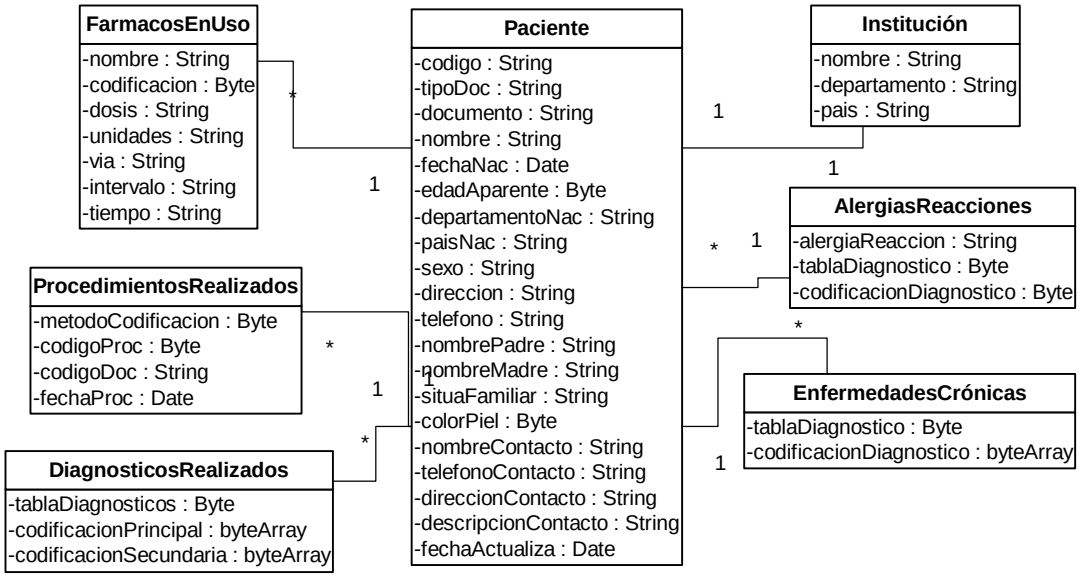


Figura 4.1

4.2.3. Características del problema

De un análisis más profundo de la estructura y características de la información que maneja el problema, se observó lo siguiente:

- La estructura de la información es conocida y está compuesta por partes estáticas (por ejemplo, el nombre del paciente) y partes dinámicas (por ejemplo, colección de fármacos)
- Presenta una estructura arborescente, donde la información se encuentra en las hojas del mismo.
- La cantidad de elementos en las diferentes colecciones depende mucho de cada paciente. Por ejemplo, puede haber un paciente sin enfermedades crónicas, pero al que se le han realizado varios procedimientos. Por lo tanto, poner un limite arbitrario, que no contemple el problema, puede traer aparejado más problemas. Además, los límites no pueden fijarse muy altos como para obviar el problema, debido a las restricciones de espacio.
- Debería disponer de un mecanismo sencillo y de bajo costo para personalizar la solución, sin necesidad de volver a reimplementar la solución a nivel de la tarjeta, el cual es el componente más crítico de la solución. El problema está orientado a trabajar con sistemas de emergencia móvil. En este momento no se cuenta con una especificación estándar de las historias clínicas resumidas a nivel nacional, por lo cual es muy probable que una aplicación de este tipo va a requerir cambios al trabajar con distintos sistemas de emergencia.
- Debería disponer de un mecanismo que permita agregar ciertos grupos de datos, por la presencia de nuevos requerimientos, esto sin necesidad de embarcar todas las tarjetas nuevamente.

4.2.4. Solución inicial

Una solución simple para este problema es realizar una representación estática de la información. Esto se logra directamente representando la jerarquía de clases que modela el problema, presentada en la figura 4.1. Este modelo debe ser reflejado tanto del lado de la tarjeta en donde se almacenaría la información, como del lado del CAD de forma de tener una representación de la información en la tarjeta a nivel del CAD.

Esta solución presenta la ventaja que es muy fácil de implementar y plantea un diseño muy cercano a la realidad que se está modelando. Por otro lado, presenta ciertas carencias que no la hacen una solución óptima, a saber:

- Realiza un manejo de memoria ineficiente. Al ser una solución estática, los tamaños de las colecciones y los tamaños de los campos individuales (como puede ser el nombre del paciente) se tienen que fijar al momento de instanciar el applet en la tarjeta[12]. Esto lleva al problema planteado en el punto 4.2.3, se tiene que fijar un tamaño arbitrario para las colecciones, no permitiendo que una colección que requiera más espacio pueda utilizar el espacio que no está siendo utilizado por otra. Este mismo manejo ineficiente se da a nivel de los campos individuales. Por ejemplo, el tamaño del string que representa el nombre tiene que ser fijo y suficientemente grande para soportar nombres largos. En la mayoría de los casos los nombres van a ser relativamente cortos, por lo cual se desaprovecha espacio.
- No está apta para evolucionar, sin tener que embarcar nuevamente la aplicación en todas las tarjetas. Cualquier agregado en los requerimientos implica cambios en el modelo de objetos representado a nivel de la tarjeta y para poder reflejar estos cambios es necesario volver a embarcar la aplicación.
- No permite personalizar la solución sin tener que reimplementar la misma. Las razones son las mismas que en el punto anterior, un cambio en los requerimientos lleva a reimplementar alguna de las clases que conforman la solución a nivel de la tarjeta.

4.3. Solución propuesta

4.3.1. Descripción general

Como se explicó en la sección anterior, la solución planteada inicialmente realiza una pobre administración de recursos, es difícil de mantener y no tiene posibilidades de evolucionar. Esto motivó la búsqueda de una solución en la cual no se presenten estos problemas.

La solución prototipada permite resolver todos los problemas antes descriptos. Ésta surge de la abstracción de la estructura de la información que se va a almacenar en la tarjeta.

Dos de las características del problema que fueron mencionadas son:

- Se puede representar mediante una estructura arborescente
- La información se encuentra almacenada solamente en la hojas del árbol

Esto sugiere que para resolver este problema es posible utilizar un árbol para estructurar y almacenar la información. Un árbol permite representar la información estática, como el nombre del paciente, en un nodo del mismo. También permite almacenar las colecciones, como un nodo que representa a la colección y los hijos del mismo que representan a los elementos de la colección. Este árbol debe permitir amoldar su estructura a la estructura de la información del problema, como se puede ver en el ejemplo de la Figura 4.2

[12] Como se señaló en la subsección 1.3.3 es aconsejable reservar toda la memoria a utilizar al momento de instanciar el applet, para evitar errores en tiempo de ejecución producidos por falta de memoria.

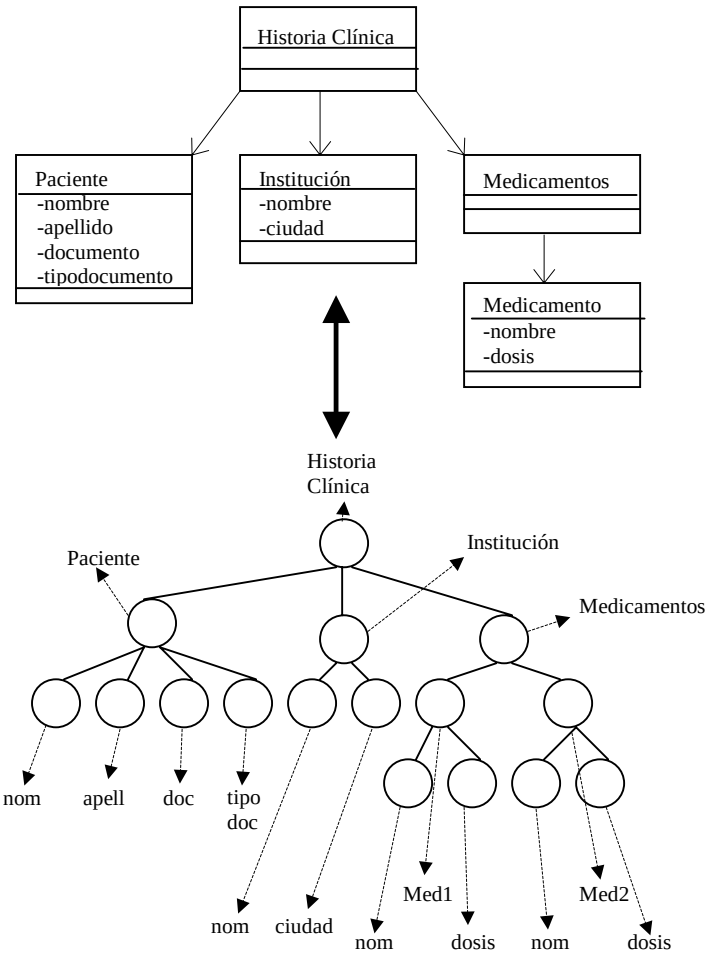


Figura 4.2

Un cambio en la estructura de la información del problema, que puede presentarse como resultado del mantenimiento de la aplicación o por la propia evolución de la realidad modelada, es asimilado fácilmente por esta solución, ya que no es necesaria su reimplementación. Basta, en estos casos, con un cambio en la estructura del árbol para incorporar el nuevo requerimiento.

Esto hace que esta solución sea muy fácil de mantener y con grandes posibilidades de evolucionar, cualidades que son de interés para la solución al problema, ya que no es necesario modificar el código para incorporar cambios, por lo cual tampoco es necesario volver a embarcar el applet en las tarjetas.

La implementación de un árbol genérico como propone esta solución, no está limitado únicamente a resolver el problema de la representación de historias clínicas, sino que permite resolver cualquier otro de similares características. Todo problema que requiera almacenar un conjunto de datos cuya estructura sea representable por un árbol con información en las hojas puede ser resuelto utilizando esta solución.

Otra característica que se observó, es que los datos que se van a manejar se pueden ver como un conjunto de parejas nombre valor, agrupados en categorías, los cuales conforman una estructura arborescente. Por ejemplo, los datos de un paciente son parte de la historia clínica y están formado un conjunto de parejas nombre valor, como se observa en la Figura 4.3

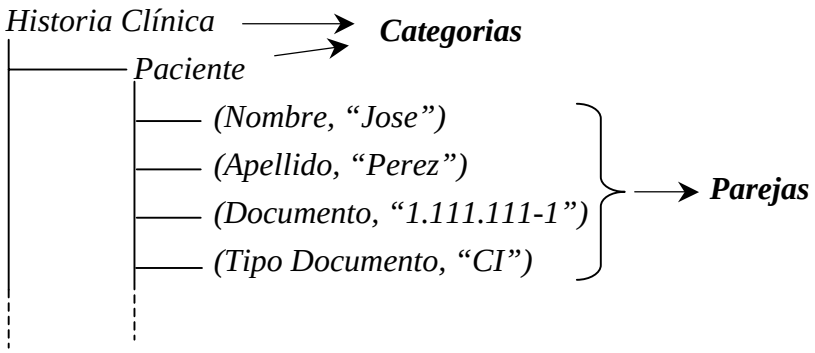


Figura 4.3

Esta característica lleva a que sea muy fácil definir la estructura de la información utilizando *Simple Markup Language (SML)*, el cual es un lenguaje de tags simple, especial para representar información con estructura arborescente. Por más información sobre *SML* ver Apéndice IV.

A continuación vemos el ejemplo de la Figura 4.3, ahora en *SML*:

```
<historiaclinica>
      <paciente>
            <nombre>Jose</nombre>
            <apellido>Perez</apellido>
            <documento>1.111.111-1</documento>
            <tipodocumento>CI</tipodocumento>
      </paciente>
      <institucion>
            <nombre>Hospital de Clinicas</nombre>
            <ciudad>Montevideo</ciudad>
      </institucion>
      <medicamentos>
            <medicamento1>
                  <nombre>Aspirina</nombre>
                  <dosis>1 por dia</dosis>
            </medicamento1>
            <medicamento2>
                  <nombre>Zolben</nombre>
                  <dosis>3 por hora</dosis>
            </medicamento2>
      </medicamentos>
</historiaclinica>
```

Esto permite que tanto la entrada como la salida de la aplicación pueda ser un archivo en SML, lo cual lleva a tener un manejo más simple y estándar de la información con la cual va a trabajar la aplicación.

4.3.2. Diseño de la solución

Como se vio en el punto anterior, esta solución contempla las carencias que tendría la representación estática de la información, pero igualmente presenta algunos problemas que es necesario resolver, a saber:

- Se debe encontrar un mecanismo para representar la estructura de un árbol dentro de la tarjeta. Dicho mecanismo deberá consumir pocos recursos.

- La estructura del árbol tendrá que ser dinámica, de forma de poder manejar las colecciones y poder realizar modificaciones al realizar el mantenimiento y evolución de la aplicación. La JCVM por el momento, no dispone de un mecanismo para la liberación de la memoria [SUN2], por lo cual la estructura del árbol deberá ser creada al momento de embarcar el applet.

- La información de los nodos se debe almacenar en forma dinámica, permitiendo optimizar el uso de los recursos de la tarjeta, ya que si se reserva un espacio fijo, éste puede ser desaprovechado o puede ser insuficiente.

El primer y segundo punto sugiere que la implementación de un árbol como tal no se debe hacer del lado de la tarjeta, primero porque esto consumiría una gran cantidad de recursos al tener que almacenar la lógica que existe detrás de un árbol y segundo por las dificultades que presenta la JCVM al momento de liberar memoria.

Por estas dos razones el árbol se va a implementar del lado del CAD y éste va a obtener la información de sus nodos, de la tarjeta. Por su parte, la tarjeta será responsable de almacenar la información de los nodos del árbol, lo cual plantea la necesidad de tener un mecanismo flexible y dinámico para manejar la información del lado de la tarjeta. La tarjeta almacenará parejas del tipo (identificador nodo, valor).

Esta necesidad junto al problema planteado en el tercer punto se resolvieron implementando un manejador de memoria propio dentro de la tarjeta, abriendo así la posibilidad de reservar y liberar memoria cuando sea necesario.

Cada vez que del lado del CAD se requiera almacenar un dato, simplemente se hace el pedido correspondiente al applet, el cual reserva la memoria necesaria y almacena los datos, utilizando el espacio necesario. El tipo de datos con el cual trabaja el manejador es el arreglo de bytes, por lo cual permite almacenar cualquier tipo que sea convertible a un arreglo de bytes.

Aún falta resolver el problema de cómo se comunica el árbol que reside en el CAD y la información almacenada en la tarjeta. Es necesario definir una codificación apropiada para poder identificar la información almacenada en la tarjeta. La codificación utilizada identifica un nodo en el árbol por su posición en el mismo.

Por ejemplo:

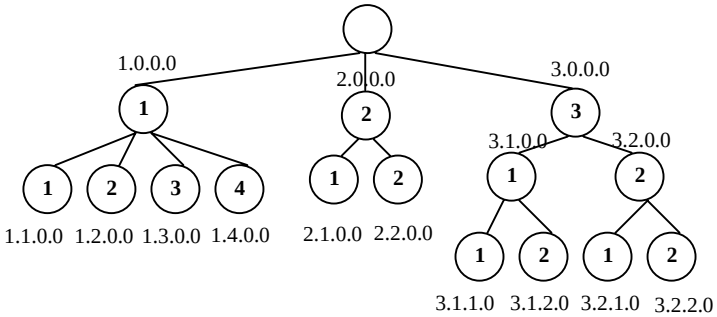


Figura 4.4

La información que está en la tarjeta no es suficiente para reconstruir el árbol del lado del CAD. Primero porque del lado del CAD es necesario saber de qué tipo son los datos de cada nodo, y segundo porque es necesario conocer qué nodos son colección ya que no es posible determinarlo con la información en la tarjeta.

Como ya fue mencionado, esta solución no sólo resuelve el problema de representar una historia clínica resumida. Cada problema que se resuelva utilizando este mecanismo, va a requerir una especificación de la estructura del árbol que define la estructura de la información almacenada en la tarjeta. Para especificar dicha estructura, se va a construir un árbol, el cual se llamará *template*. Un *template*, además de definir la estructura de la información, define el

tipo de los datos almacenados en cada hoja, qué nodos son colecciones y cuál es la estructura de los elementos de las colecciones.

La aplicación que reside en el CAD, recorre el *template*, y en base a éste y a la información almacenada en la tarjeta reconstruye el árbol. Luego la información del árbol puede ser modificada y al momento de guardar, los cambios son replicados a la tarjeta.

4.3.3. Arquitectura general

Con lo presentado hasta el momento, se puede decir que la solución consta de dos grandes componentes. Estos son:

- La aplicación que reside en el CAD, la cual tiene como objetivo representar la información del problema que se está resolviendo mediante un árbol.

- El applet del lado de la tarjeta, que va a permitir almacenar toda la información del árbol, implementando un manejador de memoria propio, de forma de optimizar la utilización de espacio dentro de la tarjeta.

Por la facilidad de mapear la información almacenada en la tarjeta a SML, se va a permitir que la aplicación en el CAD maneje indistintamente si la entrada o salida, proviene de un archivo en SML o de los datos almacenados en la tarjeta inteligente. Modelando la solución de esta manera, es posible realizar conversiones en forma natural entre lo que puede estar almacenado en una tarjeta a un archivo SML y viceversa.

En la siguiente figura se puede observar un diagrama con la arquitectura de la solución prototipada.

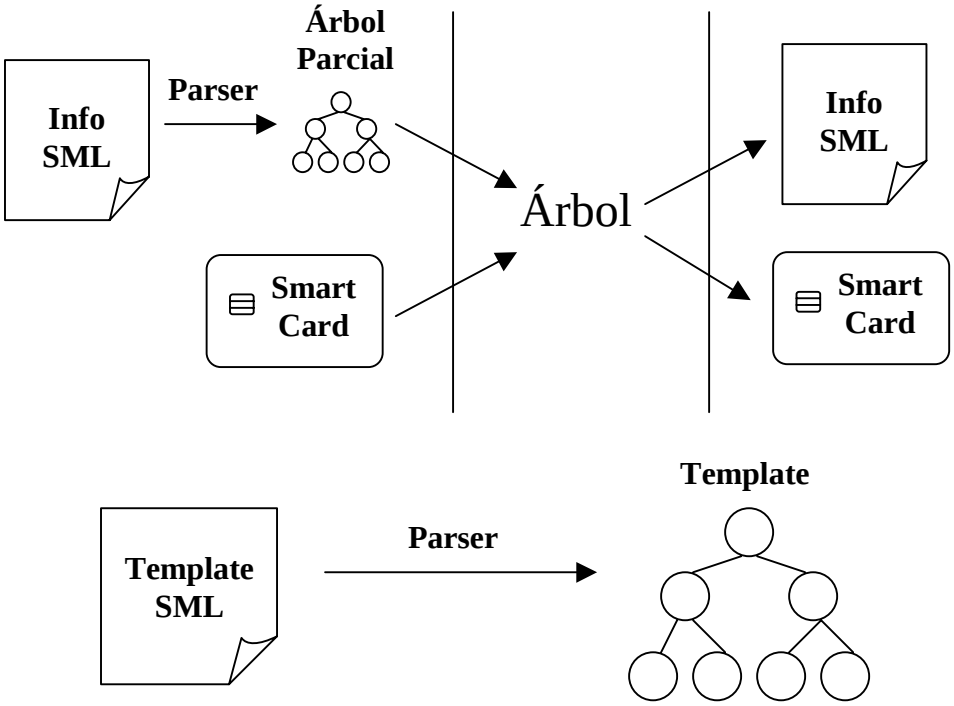


Figura 4.5

Como se ve en el diagrama, también es posible dar el template del árbol en un archivo SML.

4.3.4. Solución a nivel de la tarjeta

La responsabilidad del applet que reside en la tarjeta es la de almacenar la información de los nodos del árbol. Este va a trabajar con parejas *(identificadorNodo, valor)*, de forma que cuando se desee almacenar la información que reside en un nodo, se le debe pasar a la tarjeta el identificador del nodo y el valor a almacenar. De la misma manera cuando se requiera recuperar la información de un nodo se le pasa el identificador del nodo. Los datos que se van a almacenar pueden variar en tamaño y el applet debe permitir borrar o actualizar estos datos, pudiendo tener éstos distinto tamaño

Para ser posible esta creación, destrucción y actualización de la información del applet, es necesario disponer de un manejo flexible de la memoria. Como fue mencionado anteriormente, la JCVM no implementa un mecanismo para liberar memoria luego que ésta fue reservada [SUN2], el único mecanismo es borrando el applet de la tarjeta y volviéndolo a cargar[13].

A los efectos de solucionar este problema se implementó un manejador de memoria propio, dentro de la tarjeta. El mismo reserva un determinado espacio de memoria al momento de instanciar el applet en la tarjeta, el cual es administrado por el manejador de memoria. Éste va a permitir reservar, modificar y borrar bloques de memoria cuando sea necesario.

La memoria reservada para el manejador está destinada tanto para almacenar los datos, como para almacenar la información de control de los mismos. El manejador está compuesto principalmente por cuatro clases: *sequence*, *dinMem*, *table* y *memoryManager*, las cuales se detallan a continuación. En la figura 4.6 se puede ver el diagrama de clases simplificado del applet completo.

La clase *sequence* es la clase que está a más bajo nivel y representa una secuencia en donde el manejador almacenará todos sus datos, representados como un arreglo de bytes. Es la encargada de reservar el área de memoria que va a utilizar el manejador para almacenar sus datos. Las primitivas que posee permiten almacenar un arreglo de bytes (que de ahora en más se le llamará bloque) a partir de una determinada posición en la secuencia, así como obtener un bloque de un determinado largo, a partir de una posición en la secuencia. Esto se puede apreciar en la Figura 4.6

Un manejador de memoria como tal necesita llevar información de control de dónde está cada uno de los bloques que almacena. Esta información se almacena en estructuras de control, representadas como arreglos de bytes. Dichas estructuras, junto a los bloques de información son almacenados en la secuencia, de forma que los datos de control son almacenados en un extremo de la secuencia y los bloques de información en el otro extremo.

La clase *dinMem* es la encargada de almacenar los bloques de información dentro de la secuencia. Ésta permite tanto almacenar un bloque en la secuencia, como obtener un bloque de la misma.

Cuando se borra un bloque, el espacio de memoria asignado al mismo se pierde. Lo mismo pasa cuando se actualiza un bloque de información por otro de mayor tamaño ya que para este caso se copia el bloque a otra posición y luego se borra el bloque anterior. Esto lleva a que se produzca una fragmentación del espacio de almacenamiento. La clase *dinMem* no lleva ningún control de los espacios libres que hay, como para reutilizarlos. Eventualmente se podría extender la funcionalidad de esta clase para que permitiera la defragmentación de la secuencia, de forma de recuperar el espacio perdido por operaciones de borrado y de actualización.

[13] Siempre y cuando la implementación de la tarjeta lo permita. El kit de desarrollo utilizado para testear el caso de estudio es el Cyberflex Access de Schlumberger [SLB2] implementa esta funcionalidad.

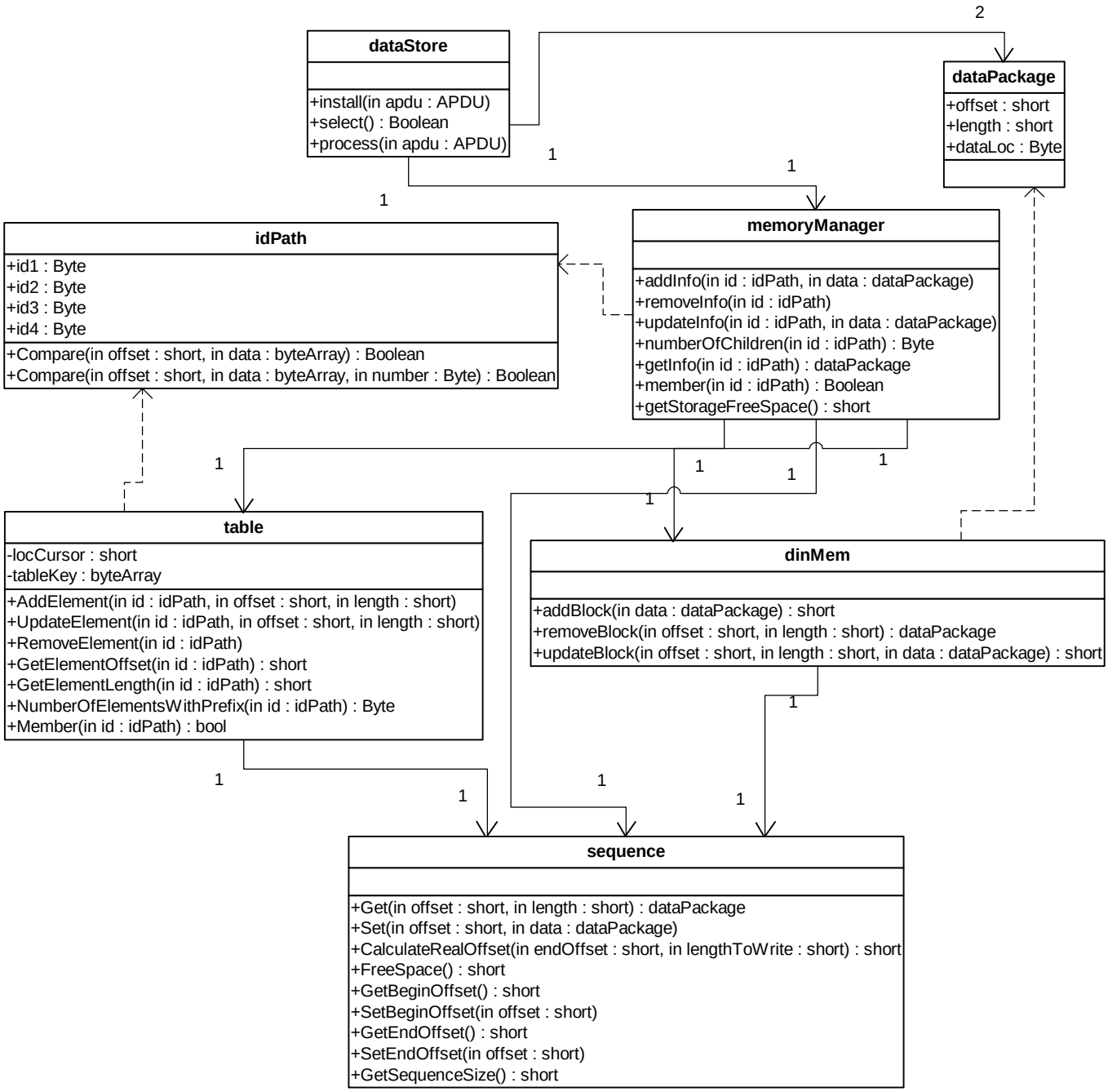


Figura 4.6

La clase *table* se encarga de llevar el registro de los bloques que se reservaron, por lo cual es la que se encarga de manejar los bloques de control. Los bloques de control están formados por un identificador del bloque (a nivel del CAD, va a ser el camino de la raíz al nodo), la posición del mismo dentro de la secuencia y su tamaño. Esto se puede ver en la Figura 4.7. Esta clase presenta métodos para agregar, borrar y actualizar bloques de control.

La clase *memoryManager* se encarga de coordinar el funcionamiento de las dos clases anteriores. Por ejemplo, cuando le llega un pedido para almacenar un determinado bloque, ésta primero reserva un área y copia el bloque a la misma, todo esto utilizando la clase *dinMem*. Luego utiliza la clase *table* para crear un nuevo bloque de control, en donde almacena el identificador del bloque junto con la posición en donde *dinMem* lo almacenó y el largo del mismo. La clase aparte de agregar bloques, permite borrar y obtener un bloque.

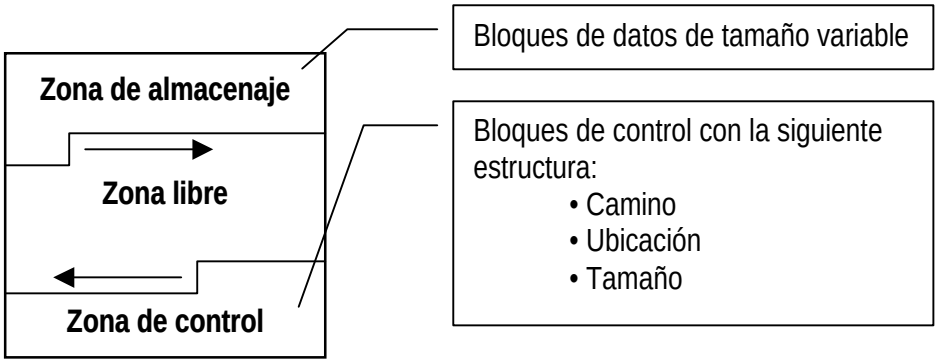


Figura 4.7

Por último se encuentra la clase *dataStore* que es el propio applet y no incorpora ninguna lógica nueva a lo ya presentado. Esta simplemente recibe los APDU's, los procesa, invoca al *memoryManager* y devuelve el resultado al CAD.

La idea inicial era resolver la comunicación entre la tarjeta y el CAD utilizando la herramienta de desarrollo **Proxy** presentada en el punto 2.2. Por problemas de tiempo y de errores en la generación automática de código no se pudo llegar a probar apropiadamente, por lo cual la comunicación se resolvió en forma manual. Para esto se implementó: del lado de la tarjeta, el método *process* encargado de interpretar los APDU's y del lado del CAD, una clase (como se verá en el punto 4.3.5) que refleja los métodos exportados por el applet. Se hizo de esta forma para minimizar los cambios a realizar al momento de migrar para utilizar el código generado por el Proxy. Actualmente se está trabajando para realizar está migración.

4.3.5. Solución a nivel del CAD

La aplicación implementada del lado del CAD tiene como principal responsabilidad la representación del árbol, el cual es el corazón de la solución propuesta. La aplicación está dividida en dos capas: capa lógica y capa de persistencia.

La capa lógica está centrada en torno a la clase *Tree*, que es un árbol n-ario, el cual posee las primitivas usuales para manejar un árbol. En la figura 4.8 se puede ver la clase *Tree* junto con el resto de las clases que conforman la capa lógica.

La clase *node* es la encargada de encapsular el comportamiento de los nodos del árbol. En la misma está almacenado el nombre del nodo, si es colección y la información del mismo. La información de cada uno de los nodos debe poder ser de tipos diferentes. El manejo de los distintos tipos será transparente para la aplicación.

Para lograr esto está la clase *Info* y todas las que heredan de ésta. La clase *Info* es una clase abstracta que define cuarto métodos abstractos, los cuales permiten el manejo de la información por las clases de arriba independientemente de los tipos. Estos métodos permiten convertir la información a un String, obtener la información parseando un string, convertir la información a un arreglo de bytes y obtener la información de un arreglo de bytes.

Para cada tipo de información que se quiera utilizar en el árbol, hay que heredar de Info y redefinir estos cuatro métodos.

La capa de persistencia es responsable de cargar y almacenar los datos del árbol, tanto desde la tarjeta inteligente como desde un archivo SML. Permite tanto cargar un árbol a partir de un template, como guardar un árbol.

Como ya se mencionó, el template brinda la estructura del árbol, el tipo de información que maneja cada nodo y marca qué nodos son colecciones. El template está representado por un árbol, en donde la información de los nodos son Strings (encapsulados por InfoString), que

contienen el nombre de una clase *Info<Tipo>* que hereda de *Info*. Estos nombres son los que determinan los tipos de información que se almacenan los nodos del árbol final.

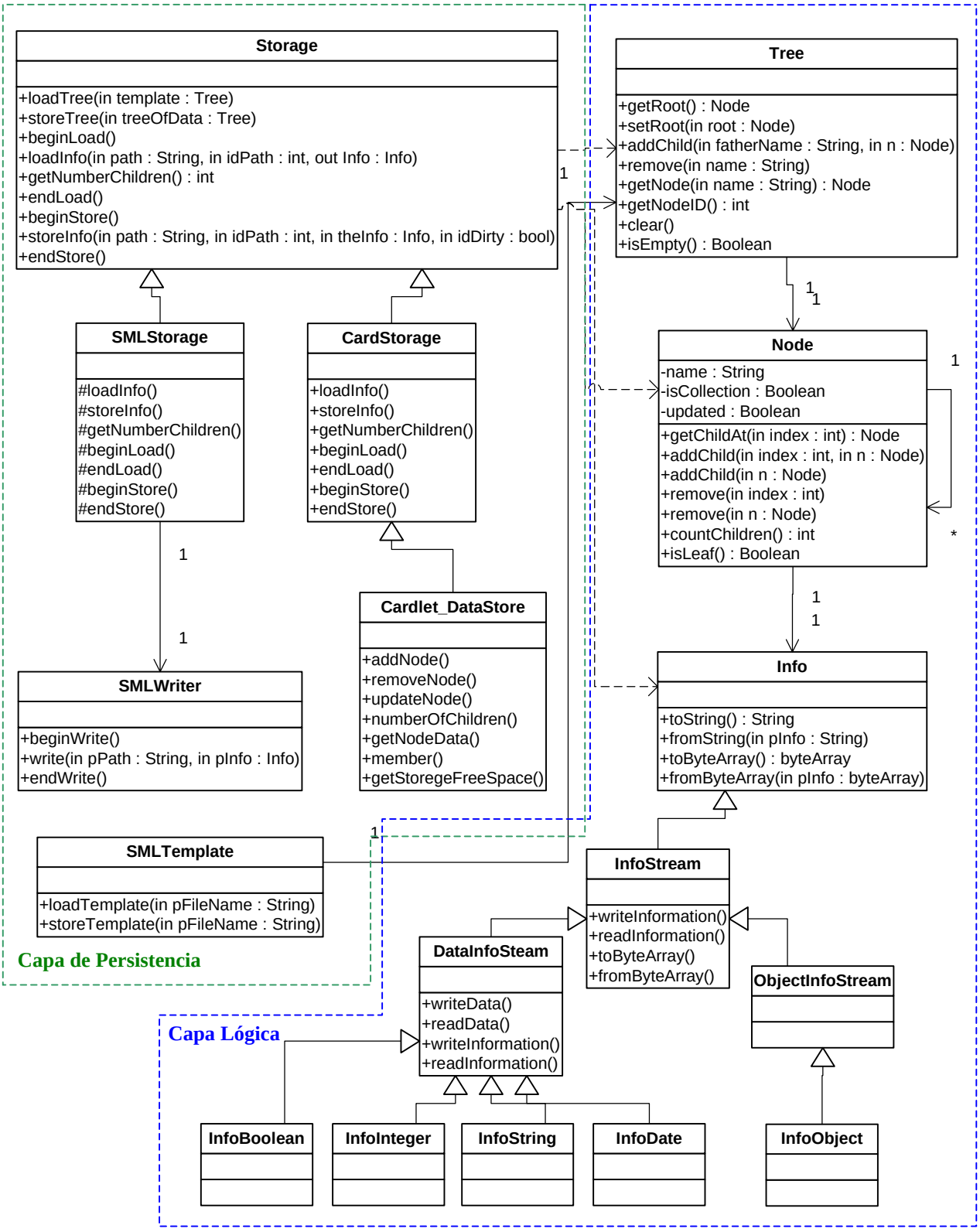


Figura 4.8

La capa de persistencia consta principalmente de tres clases (ver Figura 4.8): *Storage*, *SMLStorage* y *CardStorage*. Storage es una clase abstracta, de la cual hereda SMLStorage y CardStorage.

La clase *Storage* consta de cuatro métodos, dos de ellos, `loadTree()` y `storeTree()`, son métodos finales (no reimplementables por las clases que heredan de ella), los cuales permiten recorrer el template e ir construyendo el árbol final y recorrer un árbol y almacenarlo, respectivamente. Estos dos métodos, hacen uso de los otros dos métodos, `loadInfo()` y `storeInfo()` cuando necesitan leer la información de un nodo (en el caso de `loadTree()`) o almacenar la información de un determinado nodo (en el caso de `storeTree()`). Estos dos métodos son declarados abstractos y tienen que ser reimplementados por las clases que heredan de *Storage* y son los que determinan la forma en que la información es almacenada.

En resumen, la clase *Storage* simplemente sabe cómo a partir de un template, generar y almacenar un árbol. Pero la responsabilidad de determinar dónde y de qué forma esa información se almacena la deja a sus subclases, *SMLStorage* y *CardStorage*.

La clase *SMLStorage* hereda de *Storage*, reimplementando el método `loadInfo()`, que le permite leer una determinada información de una archivo SML y el método `storeInfo()`, que le permite almacenar una determinada información en un archivo SML.

Por otro lado la clase *CardStorage*, que también hereda de *Storage*, reimplementa los métodos para permitir leer y almacenar la información de los nodos desde la tarjeta inteligente.

Dicha clase hace uso de la clase *Cardlet_DataStore* para comunicarse con la tarjeta. Esta clase encapsula la comunicación con la tarjeta. Provee el conjunto de métodos que el applet exporta a la aplicación de terminal, de forma que del terminal solo se debe invocar un método de esta clase, la cual se va a encargar de realizar la invocación al método correspondientes en la tarjeta enviando los APDU's necesarios y posteriormente devolver la respuesta. Esto se hizo para impactar lo menos posible el código del CAD al momento de pasar a utilizar el proxy para resolver las comunicaciones. Al utilizar el proxy, éste se encargará de generar esta clase.

Los template pueden ser construidos mediante código Java, utilizando las primitivas del árbol o a partir de un archivo SML. Para el manejo de los template con SML se provee la clase *SMLTemplate*. Ésta permite generar un template a partir de un archivo SML y a partir de la representación del template como un árbol en memoria, generar un archivo SML.

4.3.6. Solución propuesta vs. solución inicial

Comparando la solución implementada con la solución propuesta inicialmente, se observa que la misma corrige todas las carencias que tenía la solución inicial, pero a su vez introduce algunas desventajas.

Las ventajas que tiene sobre la solución inicial son las siguientes:

- Realiza un manejo más eficiente de la memoria, administrando dinámicamente el área de memoria disponible. No plantea límites de tamaños en los datos, ni para la cantidad de elementos de las colecciones
- Plantea una solución completamente independiente de la estructura de los datos del problema. Esto permite que esta solución se puede reutilizar para resolver otros problemas con características similares.
- Permite que el applet evolucione para cumplir nuevos requerimientos, sin la necesidad de reimplementarlo ni de embarcar nuevamente las tarjetas.
- Permite personalizar un applet sin la necesidad reimplementarlo

Las desventajas que introduce son las siguientes:

- Tiene un nivel más de indirección. Para manipular la información siempre se debe pasar por el manejador de memoria, el cual agrega el nivel extra de indirección
- Es más difícil de implementar. Requiere implementar el manejador de memoria, mientras que la solución inicial solo requería implementar las clases del modelo de objetos, que son solamente propiedades y colecciones.
- Requiere más espacio para el código del applet. La lógica que se requiere para el manejador de memoria aumenta los requerimientos de espacio para el código. La implementación realizada ocupa aproximadamente 3.5 KB.

4.3.7. Solución aplicada a un problema general

Resumiendo lo presentado en los puntos anteriores, se dispone de una solución genérica que permite resolver problemas que involucren un conjunto de datos que presenten una estructura arborescente, y que deban ser almacenados, consultados y actualizados.

Aplicar la solución a este tipo de problemas sugiere dos etapas de desarrollo. La primer etapa consiste en definir la estructura de la información y cargar los datos en las tarjetas, y la segunda consiste en la construcción de una aplicación del lado del CAD que haga uso de los componentes descriptos anteriormente para acceder a la información.

En la primer etapa se define el template que especifica la estructura de la información que interesa almacenar. Esto se puede hacer especificando la estructura en SML y generando el árbol del template a partir de este archivo, o también creando un método en Java que construya el árbol. Es evidente que el especificar la estructura en SML es la mejor opción, ya que hacerlo en Java presenta una tarea engorrosa.

Una vez generado el árbol del template es posible obtener un árbol final, con la información propia de la instancia del problema. Para esto es necesario disponer de esta información en alguno de los medios de los cuales *Storage* permite leer, en este momento estos son de una tarjeta o de un archivo SML. Ya que el objetivo es cargar la información por primera vez a la tarjeta, se va a obtener la información del paciente de un archivo SML.

Después de generado el árbol sólo queda almacenarlo en la tarjeta inteligente, culminando así la primer etapa.

En la segunda etapa se construye una aplicación que utilice la capa lógica de la solución, para que el usuario pueda manipular la información almacenada en la tarjeta. La aplicación es la encargada de manipular el árbol, permitiendo consultar o modificar la información y cargar o guardar la información cuando sea necesario.

4.3.8. Solución aplicada al Caso de Estudio

La aplicación de la solución propuesta al caso de estudio se realizó en dos etapas, como se sugiere en la subsección anterior.

La subsección aiii.3 del Apéndice III presenta el template desarrollado. La subsección aiii.4 del mismo apéndice presenta un ejemplo de los archivos con información de los pacientes.

En la segunda etapa se desarrolló un prototipo de la aplicación donde se permite visualizar, modificar y almacenar la información de un paciente.

4.3.9. Limitaciones de la implementación

Por ser un prototipo y para simplificar la implementación del mismo, se plantearon algunas limitaciones al implementar la solución, las cuales se comentan a continuación.

- Si bien el árbol implementado del lado del CAD permite manejar árboles de cualquier longitud y cualquier cantidad de hijos, de lado de la tarjeta solamente se permiten almacenar árboles con una profundidad máxima de 4 y un máximo de 255 hijos.

- El tamaño de la información de un nodo no puede superar el tamaño de un APDU (255 bytes para información), de forma que para almacenar o recuperar la información de un nodo sea necesaria enviar un APDU.

- Las repetidas actualizaciones del contenido de la tarjeta llevan a que se produzca la fragmentación de la memoria en la tarjeta. La implementación actual no provee un mecanismo para desfragmentar la memoria de la tarjeta, por lo cual el espacio entre fragmentos es perdido.

Sección 5
Conclusiones y trabajo futuro

5.1. Conclusiones

El presente trabajo permitió obtener un panorama global del estado del arte del área de tarjetas inteligentes, en particular de JavaCards.

Se estudiaron además diversos temas relevantes en el área como ser herramientas de desarrollo, prestaciones y limitaciones de la plataforma, viabilidad de ciertos tipos de aplicación y elementos de seguridad en el desarrollo de aplicaciones.

El conocimiento adquirido permitió identificar áreas donde existen problemas y limitaciones, y desarrollar herramientas, tanto de software como metodológicas, para solucionar los mismos.

Se obtuvo experiencia práctica con dos ambientes de desarrollo para la plataforma JavaCard y se desarrollaron diversas aplicaciones, de pequeño a mediano porte. Esto hizo posible un buen entendimiento de los mismos, así como de las limitaciones mencionadas anteriormente.

Los problemas encontrados en los kit durante el desarrollo, las fallas en los mecanismos de seguridad de las tarjetas y la falta de funcionalidades de las herramientas de desarrollo sugieren que la tecnología todavía se encuentra en una etapa muy temprana de su desarrollo.

Los resultados obtenidos, y los temas que no fueron estudiados a fondo, indican que hay numerosas áreas de estudio dentro del tema tratado que deben ser atacadas con mayor profundidad.

Más allá de los resultados prácticos y conocimientos obtenidos en lo referente a la tecnología estudiada, el presente trabajo permitió a los autores iniciarse en la investigación académica y adquirir nuevas metodologías de trabajo y estudio.

Finalmente, la investigación realizada dio lugar, además, a la generación de un artículo de divulgación, y a varias charlas informativas, dos de las mismas realizadas en el marco de las VI Jornadas de Investigación Operativa e Informática del In.Co.

5.2. Trabajo futuro

Dada la variedad de temas que se trataron a lo largo del proyecto, hay varias puertas abiertas a trabajos futuros.

Como se menciona en la sección 2.2.5, el generador de código implementado puede ser mejorado en numerosos aspectos, entre los cuales se incluye:
- Incorporación y manejo de nuevos tipos de datos (particularmente strings)
- Mejora en el manejo de excepciones
- Testeo exhaustivo
- Nueva implementación con un enfoque basado en componentes (JavaBeans)

Actualmente se está trabajando en la corrección de errores en el proceso de generación de código que están limitando un testeo más profundo.

Otra extensión al generador de código que podría ser interesante es la integración al mismo de facilidades para desarrollar aplicaciones siguiendo la metodología de desarrollo propuesta en la sección 3. De esta forma, y si efectivamente es factible, se podrá integrar, en una forma más transparente, dicha metodología al desarrollo normal de applets.

En lo referente a los problemas de seguridad en object sharing, se debería realizar un caso de estudio más elaborado, y probar el mismo sobre hardware adecuado (a la fecha, no se dispone de hardware que implemente la especificación 2.1 del JCRE). Eventualmente se podría realizar también algún estudio más formal sobre la metodología propuesta, y sobre el enfoque de objetos delegados, para obtener una mejor comparación entre ambos.

Dentro del área de la seguridad, se debería incursionar en el manejo de las primitivas criptográficas soportadas por el API de JavaCard, lo cual fue dejado de lado en haras de la simplificación de los prototipos.

Un posible tema de estudio es la factibilidad de solucionar metodológicamente el problema que surge al pasar referencias como argumentos en una invocación a un método de un SIO. Como se mencionó anteriormente, esto no es posible actualmente salvo que todos los objetos que se quieran utilizar como argumentos sean SIOs, lo cual no es práctico, eficiente, ni seguro.

El prototipo de Health Card implementado puede ser mejorado en varios aspectos, como ser:

- Implementación de una interfaz de usuario más elaborada
- Incorporación de algún esquema de compactación de memoria para solucionar el problema de fragmentación externa existente en la implementación actual
- Reimplementar las clases utilizadas en la aplicación de terminal como JavaBeans para poder utilizarlas desde otras plataformas de desarrollo
- Optimización del código, y posible rediseño del código almacenado en la tarjeta para disminuir el tamaño de la aplicación.

Referencias bibliográficas

[SUN1] Java Card 2.1 Runtime Environment (JCRE) Specification, Sun Microsystems, 1999.
http://java.sun.com/products/javacard/javacard21.html

[SUN2] Java Card Virtual Machine Specification v2.1, Sun Microsystems, 1999
http://java.sun.com/products/javacard/javacard21.html

[SUN3] Java Card 2.1 API Specification, Sun Microsystems, 1999
http://java.sun.com/products/javacard/javacard21.html

[SUN4] JavaCard 2.1 Development Kit Release Notes, Sun Microsystems, 1999.
http://java.sun.com/products/javacard/javacard21.html

[SUN5] JavaCard 2.0 Language Subset and Virtual Machine Specification, Sun Microsystems, 1997.
http://www.javasoft.com/products/javacard/

[SUN6] Java Card 2.1.1 Runtime Environment (JCRE) Specification, Sun Microsystems, 1999.
http://java.sun.com/products/javacard/javacard21.html

[SUN7] The Java Language Specification - Gosling, J., Joy, B., Steele, G., Sun Microsystems, 1996
ftp://ftp.javasoft.com/docs/specs/langspec-1.0.pdf

[SUN8] The Java Virtual Machine Specification 2nd Edition - Lindholm, T., Yellin, F., Sun Microsystems, 1999
ftp://ftp.javasoft.com/docs/specs/vmspec.2nded.html.tar.gz

[ISO] ISO 7816
http://www.scia.org/aboutSmartCards/iso7816_wimages.htm

[DIGIOR] Smart Cards: a primer - Di Giorgio, R., Sun Microsystems, 1997
http://www.javaworld.com/jw-12-1997/jw-12-javadev.html

[CHAN] What is a Java Smart Card? SURPRISE 98 Report – Chan, Y.L., Chan, H.Y.

[ZHI1] Understanding Java Card 2.0 – Chen, Z., Di Giorgio, R., Sun Microsystems, 1998
http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html

[ZHI2] How to write a Java Card applet: A developer's guide – Chen, Z., Sun Microsystems, 1999
http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard_p.html

[MONKRI] Secure Object Sharing in Java Card - Montgomery, M., Krishna, K, Schlumberger Workshop on SmartCard Technology (Smartcard '99), USENIX, May 1999.

[GIR] Which Security Policy for Multiapplication SmartCards - Girard, P. Workshop on SmartCard Technology (SmartCard '99), USENIX, May 1999.

[INET1] http://www.fchn.org/
Ejemplo de un sistema de Health Card ya implementado

[INET2] http://www.healthsmartcard.net/intro.html
Ejemplo de un sistema de Health Card ya implementado

[INET3] http://www.hscaa.org/article_6.htm
Discusión de distintas aplicaciones de Health Cards

[HCR1] "Conjunto Essencial de Informaçôes do Prontuário para Integraçâo da Informaçâo em Saúde"
http://www.datasus.gov.br/prc/datasus.htm

[HCR2] Implementación de las historias clínicas en tarjetas
http://www.datasus.gov.br/cartao/datasus.htm

[AND1] Tamper Resistance - a Cautionary Note - Anderson, R., Kuhn, M., The Second USENIX Workshop on Electronic Commerce – Proceedings, 1996

[AND2] Low Cost Attacks on Tamper Resistant Devices - Anderson, R., Kuhn, M., Security Protocols, 5th International Workshop – Proceedings, 1997

[GEM] Gemplus
http://www.gemplus.com

[GX211] Gemplus GemXpressoRAD 211
http://www.gemplus.com/products/software/gemxpresso_rad_211.htm

[SLB1] Schlumberger
http://www.slb.com

[SLB2] Cyberflex Access Development Kit
http://www.cyberflex.com

[IBUT1] Dallas Semiconductor - iButton
http://www.ibutton.com

[IBUT2] http://www.ibutton.com/applications/index.html
Aplicaciones de los iButton

[FORO] http://smartie.austin.apc.slb.com/forums/cybacjavadefload/17.html

[FROST] Frost & Sullivan
http://www.frost.com

[SML1] Definiciones de conceptos del área informática
http://www.webopedia.com

[SML2] SML: Simplifying XML
Robert E. La Quey
Noviembre 24, 1999
http://www.xml.com/pub/a/1999/11/sml/index.html

[SML3] Foro de discusión de SML
http://groups.yahoo.com/group/sml-dev/

[PRV] A Secure Methodology for Object Sharing
Perovich D., Rodríguez L., Varela M.
ISSN 0797-6410
Technical Report In.Co. 00-14
2000

[W3C] World Wide Web Consortium
http://www.w3c.org

[TL] Trusted Logic
http://www.trustedlogic.com/

Apéndice I
'A Simple Methodology for Secure Object Sharing'

El trabajo descripto en la Sección 3 dio lugar a la generación de un artículo de divulgación, que ha sido publicado como reporte técnico del PEDECIBA – In.Co. [PRV], y que será sometido a una conferencia internacional a la brevedad.

El presente Apéndice presenta dicha publicación.

Las referencias de la publicación se presentan en la misma, y su codificación difiere respecto de las referencias de este documento.

A Simple Methodology for Secure Object Sharing

Daniel Perovich Leonardo Rodríguez Martín Varela
<perovich@fing.edu.uy> <lrodrigu@fing.edu.uy> <mvarela@fing.edu.uy>
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
October 2000

1 Introduction

Smart cards have been in use since the early seventies, and their applications have evolved as a result of technological advances. JavaCards are a class of smart cards that have a special Java Virtual Machine (JCVM, for JavaCard Virtual Machine) embedded. The range of applications varies from healthcare or digital wallets to loyalty programs or access control. As the JavaCard technology spreads, new applicative areas for JavaCards (and smart cards in general) are considered.
JavaCards allow more than one application (called Applets) to coexist in them. This makes them very attractive, as the user can have, for example, a banking application, an e-wallet and his driving license in the same card.

When an applet is installed, it is given an AID (Applet Identifier as defined in ISO 7816-5)[ZHI], which is unique. When a company wants to deploy an applet, it must obtain an AID for it from the ISO. In the card, applets exist in Applet Contexts, which are isolated object spaces where all the objects of a certain package coexist. The *JavaCard Runtime Environment* (JCRE) [SUN1] enforces the object space isolation by means of the Applet Firewall. The firewall prevents objects in a certain context from directly accessing objects in another context. The JavaCard specification 2.1 provides applet developers with a way to share data and services between applets. This is called object sharing.

In this paper, we put forward a methodology for secure object sharing on the JavaCard platform. Our proposal is inspired by the work of Montgomery and Krishna, from Schlumberger [MONKRI]. That work is concerned with object sharing and proposes an approach to solve some of the problems that arise in the object sharing model proposed in the JavaCard 2.1 specification [SUN1]. Their work suggests some modifications to the JCRE specification as a possible solution to those problems. We base our approach on a methodology rather than on changes to the specification.
The next section comments on the object sharing model of JavaCard 2.1 specification and on Schlumberger's approach.
In section 3 we present the proposed methodology. The case study is described in section 4. Section 5 presents some comments on our experiences with a JavaCard emulator (Sun's JCWDE), and the conclusions are presented in section 6.

2 Object Sharing

In this section we present the JavaCard 2.1 specification object sharing model.
The mechanism proposed has several flaws, which we will present, along with the solution presented in [MONKRI].

2.1 The JavaCard 2.1 Object Sharing Model

In JavaCards, each applet exists within its own *context*, along with other objects from its package, and an applet cannot directly access objects in other applet's context. This makes for increased data security, but it limits the degree of interaction between applets. In earlier specifications of the JavaCard platform, the only way for applets to share data was by means of files. These files were protected by access control lists [MONKRI]. JavaCard 2.0 specification [SUN2] introduced the concept of object sharing, but with very important restrictions [SUN3]. The JavaCard 2.1 specification introduces the notion of *Shareable Interface*, which defines a set of methods that an applet may export through the firewall. If a developer wants certain methods in his applet (the server) to be exported, he must declare them in an interface that extends the tagging interface `javacard.framework.Shareable.` This tells the JCRE that these methods can be accessed from other contexts. Then, the developer must implement the defined interface (which is called a *Shareable Interface*) in a class, and instantiate an object of that class to obtain a *Shareable Interface Object* (SIO). When another applet (client applet) wants to access the exported methods, it must first declare a reference of the type defined by the server's Shareable Interface, and then invoke the `JCSystem.getAppletShareableInterfaceObject` method, indicating the AID (application identifier) of the server applet [SUN1, SUN4] The JCRE responds to this by invoking the server applet's

`getShareableInterfaceObject`, (this method is defined in the `javacard.framework.Applet` abstract class, from which all applets descend), indicating the AID of the client applet. The server applet then decides, based on the client's AID, if the client is authorized to access the required SIO, and returns either a reference to that SIO, or null.

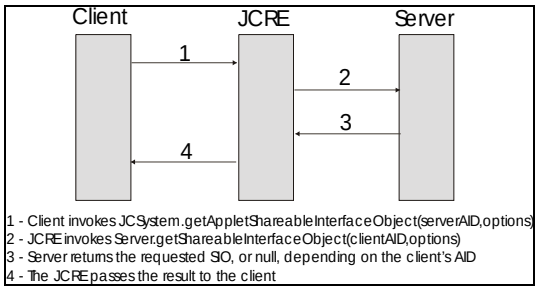


Figure ¡Error! Argumento de modificador desconocido.**. JavaCard 2.1 Object Sharing Mechanism.**

This is just an outline of the object sharing mechanism, and the reader should refer to the JCRE specification [SUN1] for details.

2.2 Problems with Object Sharing

As described in [MONKRI], the object sharing model proposed for JavaCards has serious weaknesses and limitations.

If the client's authorization to obtain a SIO is based on its AID, a malicious applet could be installed on a compromised card, with the same AID as a valid client, and thus gain access to restricted data. Although there should be security policies to prevent this kind of attack [GIR], it might not be entirely impossible to carry out. Another problem with this selection criterion is that the server applet must know the AID of every possible client, which would make impossible to allow access to new clients once the server applet has been deployed.

It is also possible for a client applet to access a SIO for which it is not authorized. This happens only if an object implements more than one shareable interface, for example A and B. When this is the case, nothing prevents a client authorized to access shareable interface A from casting the reference it gets to shareable interface B, for which it may not be authorized.

Finally, this mechanism does not allow using objects passed as parameters in a shareable method. This is because when a method declared in a shareable interface is invoked, a context switch takes place, and the firewall prevents the server applet from accessing the object received as a parameter (see [SUN1] from more details on contexts and context switching).

2.3 Schlumberger's Delegate Object approach

In [MONKRI], a solution for some of the problems that arise in the current object sharing model is presented.

The proposed solution is based on the existence of *delegate objects*. In this approach, when an applet is registered, it can also register a delegate object, which will act as its interface with other applets. This object would manage all the interaction that the applet has with other applets. Access to delegate objects would be granted to any applet requesting it, and the delegate object manages all security issues. The security mechanism proposed is the use of a secret key and a challenge/response sequence, on a per method basis. This allows two of the problems that are present in the current model to be solved: the applet impersonation problem is solved (or at least partially solved) by the challenge/response mechanism. The limitation in the number of client AIDs accepted is also solved, since any client knowing the secret key, can pass the challenge posed by the server's delegate object, and gain access to the desired methods.

Furthermore, the secret key may be different for each method or group of related methods, and thus the client can be limited to a specific set of methods, based on the secret keys that it knows.

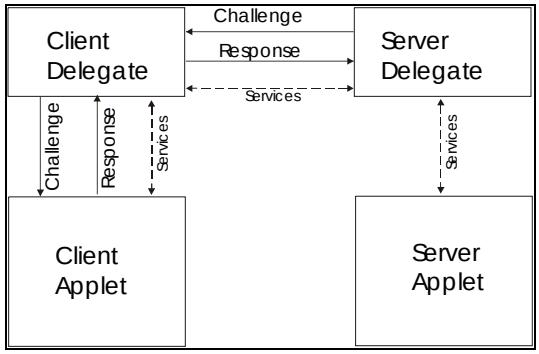


Figure ¡Error! Argumento de modificador desconocido.**. The delegate approach.**

As all shareable methods are managed by the delegate object, a client applet cannot gain unauthorized access to any method by casting, as it happens in the current model.

The only issue that remains unsolved is the impossibility of passing objects as parameters.

3 A Methodology for Secure Object Sharing

The delegate object model proposed in [MONKRI] is not the only way to solve the problems that the JavaCard 2.1 object sharing model presents.
Modifying the JavaCard specification to conform to the delegate object model is a very significant change, and it could work badly with systems based on the current specification. Some correctness criteria should be specified and verified for the proposed model, so as to make sure that the changes introduced will not affect the rest of the JCRE.
However, some of the ideas behind the delegate object approach can be implemented under the current specification, based on a development methodology, which we now proceed to present.

3.1 Overview

The basic idea behind our approach is that, in order to obtain a SIO, a client must first register itself with the server. The registration process includes an authorization process, based on a challenge/response sequence. The registration lasts, at most, for the rest of the Card Acceptance Device (CAD) session. This is to prevent the substitution of a valid client applet for a malicious one, once the client has registered itself.
Once the client is registered, it can obtain the SIOs it needs.
This methodology allows for many different implementations, which may vary depending on the security needs of the application, the number of SIOs that the server offers and so on. We will discuss some examples of this later.

3.2 Basic Components

Our methodology is based on the existence of an object in the server package, which we call *SecureSIO*. This object is an instance of an implementation of a sharable interface, called *SecureSI*. This interface, in turn, provides methods that allow a client to prove its authenticity in order to obtain the required SIOs.

The basic implementation might be improved by using an *AuthorizationManager* (AMgr), which keeps record of all registered clients together with the SIOs they can access during a session. Both a SecureSIO and an AMgr manage all the security issues within the server. A new method is added to the SecureSI by means of which a client can unregister itself after it has finished using a SIO, so as to allow the AMgr, which has limited space for registration information, to accept more entries.

3.3 How does this work?

When a client wants to obtain a SIO from a server, it invokes the method `JCSystem.getAppletShareableInterfaceObject`, as it would normally do. As mentioned in section 3.1, this JCRE method in turn invokes the server's `getShareableInterfaceObject` method. This method should be redefined in the server to act as follows: the server queries the AMgr to verify that the client is authorized to obtain the SIO it is asking for. If the client is authorized, the server then returns the corresponding SIO. Otherwise, the server returns the SecureSIO, to allow the client to register itself. The client then proves its authenticity, and asks again for the desired SIO.

To obtain authorization, the client must ask the SecureSIO for a challenge, which will eventually depend on the SIO it is asking for. Once the client gets the challenge, it must provide a response for it, and if that response is correct, the SecureSIO proceeds to register it with the AuthorizationManager. It is important to note that in every moment, it is the client that initiates the communication with the server. This is to prevent a malicious applet from posing the client different challenges, and using it as a translator.

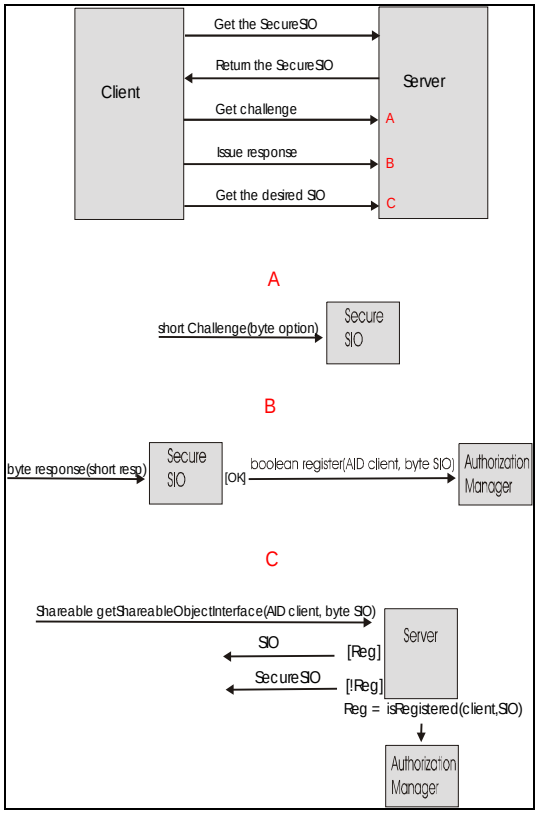


Figure ¡Error! Argumento de modificador desconocido.**. The proposed Object Sharing Mechanism.**

Making the client start all communications does not make the hack mentioned above completely impossible, but it makes it harder to implement, since the client must be externally stimulated to make it ask for a certain SIO, and then use it to translate a secret key.
Given that the number of registered clients in the AuthorizationManager is limited, and in some cases, there could be many applets interacting with the server, the client may unregister itself to free space in the list of registered clients. In order to do this, it requires the SecureSIO to the server, and uses the `unregister` method it provides.
We now proceed to comment on some implementation issues that must be considered for the system to work as desired.

3.4 Applet Impersonation and Casting

Inappropriate casting is feasible only when several shareable interfaces are implemented in a single class. A way around this problem is to implement each shareable interface on a separate class, and to make sure these classes are not in the subclassing relation. This makes it impossible for the client to cast the SIO it obtains to another shareable interface [MONKRI].

As to impersonation, the AMgr is required to store the session's authorizations in CLEAR_ON_RESET transient objects, so that a client that has been registered during a session cannot be replaced with a malicious one for the next CAD session.

In addition, the AuthorizationManager can be implemented so that it stores authorizations for clients at method level, thus achieving the same granularity that delegate objects provide.

4 A Small Case Study

We now turn to present an experiment we developed using the *JavaCard Workstation Desktop Environment* (JCWDE), SUN's JavaCard 2.1 emulator.

We implemented a small and very simple server, which offers two different SIOs, and interacts with a number of clients. The server applet itself does nothing but receive requests from the clients. All the services offered are implemented in the SIOs, which reduces the server's logic to its `getShareableInterfaceObject` method.
The two SIOs offered are a small wallet, which exports three methods and a simplified medical record, which exports two methods.

The interaction between the server and the clients is as depicted in figure 3. When the client wants a certain SIO, it asks the server for it. The implementation of the server's

```
public Shareable
getShareableInterfaceObject(AID client,
byte sio){
       switch(sio){
            case SECURE_SIO:
                          return sSIO;

            case SHAREABLE_POCKET: {
                          if
(mgr.isRegistered(client,sio)){
                             return pocket;
                          }
                          else{
                             return sSIO;
                          }

            }
            case SHAREABLE_DATA: {

  if(mgr.isRegistered(client,sio)){
                             return data;
                          }
                          else{
                             return sSIO;
                          }

            }
            default: return null;
       }

}
```

Figure 4. Server's getShareableInterfaceObject method.

`getShareableInterfaceObject` method enforces the mechanism proposed, by checking with the AMgr (mgr) to verify whether the client is registered for that SIO or not.

If the client is registered, the method returns the appropriate SIO (be it data, or pocket). In case the client is not registered, it returns the SecureSIO (sSIO). In this implementation, the method returns `null` if an invalid SIO is asked for.

In this small example, the security issue is outlined, but not properly solved. The SecureSIO does not use any cryptographic protection, as it should. We left it out in order to keep the example simple.
When a client invokes the `register` method on the SecureSIO, it can obtain three possible results, depending on both the response provided and the space available in the Authorization Manager. Although the methodology allows for many clients requesting many SIOs from a single server in a single CAD session, most applications probably won't use that much applet interaction, and so the AMgr's space limitations shouldn't be an issue.

Figure 5. The SecureSIO class.

```
public class SecureSIO implements SecureSI {
   short currentChallenge = -1;
   byte currentSIO = 0;
   private AuthorizationManager mgr = null;


   public SecureSIO(AuthorizationManager amgr){
      mgr = amgr;
   }


   public short challenge(byte sio){
      currentSIO = sio;
      currentChallenge = sio;
      return currentChallenge;

   }

   public byte response(short resp){
      if(responseOk(resp)){

   if(mgr.register(JCSystem.getPreviousContextAID(),curre
ntSIO)){
            return SecureSI.RESPONSE_OK;
         }
         else{
            return SecureSI.NO_ROOM;
         }
      }
      else{
         return SecureSI.RESPONSE_FAILED;
      }

   }

   public void unregister(byte sio){
      mgr.unregister(JCSystem.getPreviousContextAID(),
sio);
   }


   private boolean responseOk(short resp){

      return (resp == currentChallenge);

   }
}
```

The implementation of the AuthorizationManager consists of a pair of arrays, containing a list of client AID's, and a list of SIOs, which are managed in parallel. Each pair of elements represents an entry to the AuthorizationManager.

```
public class AuthorizationManager {
   private Object[] currentClients = null;
   private byte[] authorizedSIOs = null;
   private byte currentAmount = 0;

protected AuthorizationManager(byte amount)
throws SystemException {
     byte i;
     currentClients =
JCSystem.makeTransientObjectArray(amount,
JCSystem.CLEAR_ON_RESET);
     authorizedSIOs =
JCSystem.makeTransientByteArray(amount,
JCSystem.CLEAR_ON_RESET);
     for(i = 0;i< currentClients.length;
i++){
        currentClients[i] = null;
     }
   }

public boolean isRegistered(AID aid, byte
sio){
     byte i = 0;
     while(i<currentAmount){
        if (aid.equals(currentClients[i]) &&
sio==authorizedSIOs[i]){
           return true;
        }
        else {
           i++;
        }
     }
     return false;
   }

public void unregister(AID aid, byte sio){
     byte i = 0;
     while(i<currentAmount){
        if (aid.equals(currentClients[i]) &&
authorizedSIOs[i++] == sio){
           authorizedSIOs[i] =
authorizedSIOs[currentAmount - 1];
           currentClients[i] = currentClients[-
-currentAmount];
           break;
        }
     }
}

public boolean register(AID aid, byte sio){
   if (isRegistered(aid,sio)) return true;
     if(currentAmount==currentClients.length)
     {return false;
     }
     else{
        authorizedSIOs[currentAmount] = sio;
        currentClients[currentAmount++] = aid;
        return true;
     }
   }

}
```

Figure 6. The AuthorizationManager class.

5 Notes on our experience with Sun's JCWDE

All the implementations that we have done so far have been developed with the Sun's JavaCard Development Kit 2.1. This kit provides tools for converting and verifying class files, emulating a JavaCard, and testing applets with APDU scripts. As to simulation itself, it provides the JCWDE, which is a (limited) card simulator, and the APDUtool, which is an APDU scripting tool.

Based on our experience with this kit, we believe that it needs some improvements. Firstly, and as described in the JavaCard Development Kit Release Notes [SUN5], there are core aspects of a JavaCard that have been left out, such as the firewall, and the impossibility of simulating a card reset.

Secondly, there is no easy way of creating APDU scripts, which must be written as strings of hex numbers. This could be easily improved, and it would save a great deal of time and errors.

6 Conclusions

We presented a methodology that may help developers to avoid some of the problems that arise when using the JavaCard 2.1 object sharing model.

The methodology proposed is simple to implement, and very flexible.

Although the overhead introduced by the use of the methodology may be greater than that of the delegate object approach, it does not require any changes in the JavaCard specification, which allows developers to use it with the existing implementations of JavaCard 2.1.

Since we only tested the methodology on a very simple case, and running on an emulator, there is still a lot of work to do, starting by implementing it on a real system, and on more complex cases.

7 References

- [MONKRI] Montgomery, M. and Krishna, K. - Secure Object Sharing in Java Card. Workshop on Smartcard Technology (Smartcard '99), USENIX, May 1999.

- [ZHI] Zhiqun Chen - How to write a Java Card Applet: A developer's guide http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html

- [SUN1] Java Card 2.1 Runtime Environment (JCRE) Specification. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [SUN2] JavaCard 2.0 Language Subset and Virtual Machine Specification. Sun Microsystems, 1997. http://www.javasoft.com/products/javacard/

- [SUN3] JavaCard 2.0 Programming Concepts. Sun Microsystems, 1997. http://www.javasoft.com/products/javacard/

- [SUN4] JavaCard 2.1 Application Programming Interface. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [SUN5] JavaCard 2.1 Development Kit Release Notes. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [GIR] Girard, P. - Which Security Policy for Multiapplication SmartCards Workshop on SmartCard Technology (SmartCard '99), USENIX, May 1999.

8 Acknowledgements

We would like to thank all the people who helped us in the process of writing this paper, especially Eduardo Giménez and Gustavo Betarte.

Apéndice II
Ejemplo de utilización del Proxy

aii.1. Introducción

El presente apéndice ejemplifica el uso del generador de código desarrollado. Se introduce a continuación el problema a resolver. Las siguientes subsecciones muestran la implementación y la corrida del Proxy. En último lugar se muestra el código generado por la herramienta desarrollada. Como se menciona en el documento principal, actualmente se continúa el desarrollo de esta herramienta, debido a que todavía presenta algunos errores en el código generado.

aii.2. Descripción de la realidad

El ejemplo seleccionado es muy simple, pero permite ilustrar los conceptos básicos de la utilización del generador de código y del proxy generado por el mismo. Se verá la implementación de un monedero electrónico (ePurse).

Dicho monedero provee métodos para incrementar, decrementar y consultar el saldo disponible.

El siguiente diagrama presenta el diseño del monedero electrónico:

MiPurse
- amount : short
+ getAmount() : short + credit(in pAmount : short) + debit(in pAmount : short)

aii.3. Implementación

A continuación se expone el código del applet. Cabe notar que es una versión muy simplificada de un ePurse, ya que los controles que se realizan son mínimos, así como las funcionalidades que ofrece.

```
//#AID 0x09 0xEE 0x00 0x05 0xA0 0x40 0x30 0x03
//#EXC 1111 UpperBoundException
//#EXC 2222 LowerBoundException



package test;

import javacard.framework.*;


public class MiPurse extends Applet {

  private short count;


  protected MiPurse()
  {
     count=(short)0;
     register();
  }



  public boolean select() {
    return true;
  }

  public static void install(APDU apdu){
      new MiPurse();
  }




  //#CAD
  public void credit (short cred){

    if((short)(Short.MAX_VALUE - count) >= cred){

      count = (short)(count + cred);

    }else{

      UserException.throwIt((short)0x1111);

    }


  }

  //#CAD
  public void debit (short deb){

    if((short)(count - deb) < 0 ){

      count = (short)(count - deb);

    }else{

    UserException.throwIt((short)0x2222);

    }


  }

  //#CAD
  public short getCount (){
    return count;
  }

}
```

En el código se puede apreciar la utilización de comentarios con un formato especial (*//#AID*) para indicar el AID de la aplicación, otro formato (*//#EXC*) para declarar un mapeo entre las excepciones que lanza el applet (recordar que las excepciones que salen de la tarjeta sólo se indican con un código de error) con las que lanzará el proxy a la aplicación de terminal, y un tercer formato (*//#CAD*) que indica los métodos que deberán ser ofrecidos por el proxy.

El uso de casts en varias partes del código se debe a que la JCVM está limitada en los tipos primitivos, que maneja, por lo que los resultados de operaciones como la suma, que son de tipo *int* (32 bits) deben ser casteados a tipo *short* (16 bits)

aii.4. Ejecución del Proxy

El proceso de generación, utilizando el prototipo actual consta de dos pasos:

1- Se debe ejecutar el generador sobre el código del applet. Actualmente esto se realiza como sigue:
   `java -cp . proxy.Correr MiPurse true true < MiPurse.java`
   Esto indica que se va a generar un proxy para la clase MiPurse, que se deberá manejar las excepciones (primer "true") y que se espera salida estilo "verbose" (segundo "true").
2- Se inserta el código generado para el método process y métodos auxiliares en el código del applet

aii.5. Código generado

aii.5.1. Código para el CAD

El código que se genera para la aplicación de terminal es el del proxy propiamente dicho.

```
//Proxy for applet MiPurse

//by ProxyDev 0.1


package proxies;

import javacard.framework.*;
import t5.sccad.*;


  import java.util.*;


public class MiPurseProxy extends ProxyCardlet{


  private Vector exceptions;

  private static int[] temp = { 9,  238,  0,  5,  160,  64,  48,  3};
  private static t5.sccad.AID proxyAID = new t5.sccad.AID(temp);


  public MiPurseProxy(SmartCard sc) {

    super(sc, proxyAID);
  exceptions = new Vector();

  exceptions.add("1111");
  exceptions.add("UpperBoundException");
  exceptions.add("2222");
  exceptions.add("LowerBoundException");
```

```
  }


public void credit(short cred)throws Exception {

  Result res = null;
  Invocation inv = new Invocation(proxyAID, (byte) 0);
    inv.addParam(cred);

    res = send(inv);
  if(res.exceptionThrown()){
    String code = Byte.valueOf(String.valueOf(res.getSW1()),16).toString() +
Byte.valueOf(String.valueOf(res.getSW2()),16).toString();
    Exception e;
    int i = exceptions.indexOf(code);
    if(++i > 0){
      e = (Exception)
Class.forName(exceptions.elementAt(i).toString()).newInstance();
    }
    else{
      e = new Exception();
    }
      throw e;
  }
    return;  }

public void debit(short deb)throws Exception {

  Result res = null;
  Invocation inv = new Invocation(proxyAID, (byte) 1);
    inv.addParam(deb);

    res = send(inv);
  if(res.exceptionThrown()){
    String code = Byte.valueOf(String.valueOf(res.getSW1()),16).toString() +
Byte.valueOf(String.valueOf(res.getSW2()),16).toString();
    Exception e;
    int i = exceptions.indexOf(code);
    if(++i > 0){
      e = (Exception)
Class.forName(exceptions.elementAt(i).toString()).newInstance();
    }
    else{
      e = new Exception();
    }
      throw e;
  }
    return;  }

public short getCount()throws Exception {

  Result res = null;
  Invocation inv = new Invocation(proxyAID, (byte) 2);

    res = send(inv);
  if(res.exceptionThrown()){
    String code = Byte.valueOf(String.valueOf(res.getSW1()),16).toString() +
Byte.valueOf(String.valueOf(res.getSW2()),16).toString();
    Exception e;
    int i = exceptions.indexOf(code);
    if(++i > 0){
      e = (Exception)
Class.forName(exceptions.elementAt(i).toString()).newInstance();
    }
    else{
      e = new Exception();
    }
      throw e;
  }
    return res.getShort();
  }

public t5.sccad.AID getAID(){

  return proxyAID;
}
```

```
}
```

aii.5.2. Código para la tarjeta

El código generado para la tarjeta incluye el método process del applet, así como algunos métodos y declaraciones auxiliares. Los métodos auxiliares que deben ser incluidos son deducidos a partir del código del applet, de forma de sólo incluir el código necesario, disminuyendo de esta forma el tamaño del código a cargar en la tarjeta.

```
    private short _proxy_count = ISO.OFFSET_CDATA;

  private boolean _proxy_incoming = false;

  private short _proxy_getShortParam(APDU apdu){

    if(!_proxy_incoming){
      byte byteRead = (byte)(apdu.setIncomingAndReceive());
      _proxy_incoming = true;
    }

    return (short)((short)(apdu.getBuffer()[_proxy_count++]<<(byte)8) +
apdu.getBuffer()[_proxy_count++]);
  }

  private void _proxy_setShort(APDU apdu, short data){
    apdu.getBuffer()[ISO.OFFSET_CDATA] = (byte)(data >> 8);
    apdu.getBuffer()[(short)(ISO.OFFSET_CDATA+1)] = (byte)(data &
(short)255);
  }

  public void process(APDU apdu){

  if(((apdu.getBuffer()[ISO.OFFSET_CLA] == (byte)0xF0) ||
(apdu.getBuffer()[ISO.OFFSET_CLA] == (byte)0)) &&
(apdu.getBuffer()[ISO.OFFSET_INS] == (byte) 0xA4))
    {
      ISOException.throwIt(ISO.SW_NO_ERROR);
    }
    _proxy_incoming = false;        // reset recieving state
    _proxy_count = ISO.OFFSET_CDATA; //reset global counter

    short s = 0;

    switch((byte)(apdu.getBuffer()[ISO.OFFSET_INS])){
      case 2: {
        try{
        credit(_proxy_getShortParam(apdu));

        return;


        } catch(UserException e0){

           ISOException.throwIt(e0.getReason());

        }
      }

      case 4: {
        try{
        debit(_proxy_getShortParam(apdu));

        return;


        } catch(UserException e1){

           ISOException.throwIt(e1.getReason());

        }
      }
```

```
      case 6: {
        try{
        s = getCount();

        _proxy_setShort(apdu,s);
        apdu.setOutgoingAndSend(ISO.OFFSET_CDATA,(short)2);
        return;


        } catch(UserException e2){

            ISOException.throwIt(e2.getReason());

        }
      }
    }

      return; //should never reach this code

  }
```

Apéndice III
Detalles del Caso de Estudio

aiii.1. Introducción

El presente apéndice introduce información detallada referente al caso de estudio. Las siguientes subsecciones muestran el estudio de tamaños de la información a almacenar, el documento SML que representa el template de los datos utilizados, y un archivo de ejemplo.

aiii.2. Estudio del tamaño de la información

El estudio realizado acerca del tamaño de la información a almacenar presenta las categorías de interés, así como cada ítem dentro de la categoría. Se indica el tipo de datos y el tamaño de cada uno de los ítems. Presenta la cantidad de ítems necesarios así como el tamaño total. Para reducir el tamaño de alguno de los ítems se utilizó cierta codificación, la cual se indica en la última columna.

Categoría	Item	Tipo de datos	Tamaño (bytes)	Codificación
Datos administrativos	Código del paciente	char	10	
	Tipo de documento	char	3	CI | DNI | PS
	Documento	char	15	
	Nombre	char	50	
	Fecha de nacimiento	date	8	
	Edad aparente	byte	1	
	Departamento de nacimiento	char	2	
	País de nacimiento	char	2	UY | AR | …
	Sexo	char	1	F | M | I | U
	Dirección	char	100	
	Teléfono	char	20	
	Nombre del padre	char	50	
	Nombre de la madre	char	50	
	Situación familiar	char	2	
	Color de piel	byte	1	
	Nombre del contacto	char	50	
	Teléfono del contacto	char	20	
	Dirección del contacto	char	100	
	Descripción del contacto	char	50	
	Fecha de última actualización	date	8	
Subtotal para esta categoría			543	
Datos de la institución de salud	Nombre de la institución	char	100	
	Departamento	char	2	
	País	char	2	

	Subtotal para esta categoría		104	
Alergias y reacciones adversas	Alergia o reacción	char	100	
	Tabla de diagnóstico	byte	1	
	Codificación de diagnóstico	byte	10	
	Subtotal para esta categoría (10 entradas)		1110	
Enfermedades crónicas	Tabla de diagnóstico	byte	1	
	Codificación del diagnóstico y dolencias crónicas	byte	10	
	Subtotal para esta categoría (10 entradas)		110	
Diagnósticos realizados	Tabla de diagnósticos utilizada	byte	1	
	Codificación de diagnóstico principal	byte	10	
	Codificación de diagnóstico secundaria	byte	10	
	Subtotal para esta categoría (10 entradas)		210	
Procedimientos realizados	Método de codificación de procedimientos	byte	1	
	Código del procedimiento	byte	10	
	Nombre o código del profesional que lo realizó	char	50	
	Fecha del procedimiento	date	8	
	Subtotal para esta categoría (10 entradas)		690	
Fármacos en uso	Nombre o denominación	char	50	
	Codificación del fármaco	byte	10	
	Dosis	char	100	
	Unidades	char	5	
	Vía	char	10	
	Intervalo	char	25	
	Tiempo de utilización	char	25	
	Subtotal para esta categoría (10 entradas)		2250	
	Tamaño Total		5017	

Tabla III.1

El tamaño calculado es de aproximadamente 5 KB en total. El tamaño es relativamente adecuado para ser almacenado en una tarjeta.

Notar, sin embargo, las limitaciones en los tamaños de cada ítem, que lleva a desperdiciar espacio en algunos casos y a quedar escaso en otros. Un ejemplo claro es el campo nombre, donde la mayoría de los nombres tienen menos de 50 caracteres, pero hay casos en que más caracteres son necesarios.

Otro inconveniente es que se asume 10 entradas en las categorías donde se aceptan más de una entrada. Las mismas consideraciones que antes se aplican aquí.

Por último, se debe notar que algunos ítems sólo almacenan el código, lo que implica una universalización de la codificación, que no hay.

aiii.3. Archivo template.sml

A continuación se lista el template utilizado para representar la información de la historia clínica resumida.

```
<HistoriaClinica>
        <DatosAdministrativos>
                <Codigo>InfoString</Codigo>
                <Documento>
                        <Tipo>InfoString</Tipo>
                        <Numero>InfoString</Numero>
                </Documento>
                <NombreCompleto>
                        <Titulo>InfoString</Titulo>
                        <Apellido>InfoString</Apellido>
                        <IncialSegundoNombre>InfoString</IncialSegundoNombre>
                        <Nombre>InfoString</Nombre>
                        <Sufijo>InfoString</Sufijo>
                </NombreCompleto>
                <FechaNacimiento>InfoDate</FechaNacimiento>
                <EdadAparente>InfoByte</EdadAparente>
                <LugarNacimiento>
                        <Localidad>InfoString</Localidad>
                        <Pais>InfoString</Pais>
                </LugarNacimiento>
                <Sexo>InfoString</Sexo>
                <ColorDePiel>InfoString</ColorDePiel>
                <Telefono>InfoString</Telefono>
                <Direccion>
                        <Calle>InfoString</Calle>
                        <CP>InfoString</CP>
                        <Localidad>InfoString</Localidad>
                        <Pais>InfoString</Pais>
                </Direccion>
                <Familia>
                        <Padre>InfoString</Padre>
                        <Madre>InfoString</Madre>
                        <Otro>InfoString</Otro>
                        <SituacionFamiliar>InfoString</SituacionFamiliar>
                </Familia>
                <Contacto>
                        <Nombre>InfoString</Nombre>
                        <Descripcion>InfoString</Descripcion>
                        <Telefono>InfoString</Telefono>
                        <Direccion>
                                <Calle>InfoString</Calle>
                                <CP>InfoString</CP>
                                <Localidad>InfoString</Localidad>
                                <Pais>InfoString</Pais>
                        </Direccion>
                </Contacto>
                <FechaUltimaActualizacion>InfoDate</FechaUltimaActualizacion>
        </DatosAdministrativos>
        <DatosInstitucion>
                <Nombre>InfoString</Nombre>
                <Telefono>InfoString</Telefono>
                <Direccion>
                        <Calle>InfoString</Calle>
                        <CP>InfoString</CP>
                        <Localidad>InfoString</Localidad>
                        <Pais>InfoString</Pais>
                </Direccion>
        </DatosInstitucion>
        <Alergias*>
                <Alergia>
                        <Descripcion>InfoString</Descripcion>
                        <TablaDiagnostico>InfoString</TablaDiagnostico>
                        <CodigoDiagnostico>InfoString</CodigoDiagnostico>
                </Alergia>
        </Alergias>
```

```
        <EnfermedadesCronicas*>
                <EnfermedadCronica>
                        <Descripcion>InfoString</Descripcion>
                        <TablaDiagnostico>InfoString</TablaDiagnostico>
                        <CodigoDiagnostico>InfoString</CodigoDiagnostico>
                </EnfermedadCronica>
        </EnfermedadesCronicas>
        <DiagnosticosRealizados*>
                <DiagnosticoRealizado>
                        <Descripcion>InfoString</Descripcion>
                        <TablaDiagnostico>InfoString</TablaDiagnostico>
                        <CodigoDiagnostico>InfoString</CodigoDiagnostico>
                        <CodigoDiagnosticoSecund>InfoString</CodigoDiagnosticoSecund>
                </DiagnosticoRealizado>
        </DiagnosticosRealizados>
        <ProcedimientosRealizados*>
                <ProcedimientoRealizado>
                        <Descripcion>InfoString</Descripcion>
                        <MetodoCodificacion>InfoString</MetodoCodificacion>
                        <Codigo>InfoString</Codigo>
                        <NombreProfesional>InfoString</NombreProfesional>
                        <Fecha>InfoDate</Fecha>
                </ProcedimientoRealizado>
        </ProcedimientosRealizados>
        <FarmacosEnUso*>
                <Farmaco>
                        <Nombre>InfoString</Nombre>
                        <Codigo>InfoString</Codigo>
                        <Dosis>
                                <Unidades>InfoString</Unidades>
                                <Via>InfoString</Via>
                                <Intervalo>InfoString</Intervalo>
                        </Dosis>
                        <Desde>InfoDate</Desde>
                        <TiempoUtilizacion>InfoString</TiempoUtilizacion>
                </Farmaco>
        </FarmacosEnUso>
</HistoriaClinica>
```

aiii.4. Archivo SML de ejemplo

A continuación se presenta un ejemplo de una historia clínica resumida del paciente Juan Pérez basada en el template anterior.

```
<HistoriaClinica>
        <DatosAdministrativos>
                <Codigo>70110</Codigo>
                <Documento>
                        <Tipo>CI</Tipo>
                        <Numero>1.475.214-6</Numero>
                </Documento>
                <NombreCompleto>
                        <Titulo>Arq.</Titulo>
                        <Apellido>Perez</Apellido>
                        <Nombre>Juan</Nombre>
                </NombreCompleto>
                <FechaNacimiento>4/5/1942</FechaNacimiento>
                <LugarNacimiento>
                        <Localidad>Montevideo</Localidad>
                        <Pais>UY</Pais>
                </LugarNacimiento>
                <Sexo>M</Sexo>
                <ColorDePiel>Blanco</ColorDePiel>
                <Telefono>555-5555</Telefono>
                <Direccion>
                        <Calle>Gral. Flores 1223</Calle>
                        <Localidad>Montevideo</Localidad>
                        <Pais>UY</Pais>
                </Direccion>
                <Familia>
                        <Padre>Esteban Perez</Padre>
                        <Madre>Raquel Gonzalez</Madre>
```

```
            </Familia>
            <Contacto>
                <Nombre>Pablo Perez</Nombre>
                <Descripcion>Hijo</Descripcion>
                <Telefono>555-1234</Telefono>
                <Direccion>
                    <Calle>Av. Italia 2321</Calle>
                    <Localidad>Montevideo</Localidad>
                    <Pais>UY</Pais>
                </Direccion>
            </Contacto>
            <FechaUltimaActualizacion>3/11/2000</FechaUltimaActualizacion>
        </DatosAdministrativos>
        <DatosInstitucion>
            <Nombre>Hospital de Clinicas</Nombre>
            <Direccion>
                <Calle>Av. Italia</Calle>
                <Localidad>Montevideo</Localidad>
                <Pais>UY</Pais>
            </Direccion>
        </DatosInstitucion>
        <Alergias>
            <Alergia>
                <Descripcion>Iodo</Descripcion>
            </Alergia>
        </Alergias>
        <DiagnosticosRealizados>
            <DiagnosticoRealizado>
                <Descripcion>Dolor de cabeza</Descripcion>
            </DiagnosticoRealizado>
        </DiagnosticosRealizados>
        <ProcedimientosRealizados>
            <ProcedimientoRealizado>
                <Descripcion>Fractura de radio</Descripcion>
                <NombreProfesional>Dr. Schlum Berger</NombreProfesional>
                <Fecha>3/9/1991</Fecha>
            </ProcedimientoRealizado>
            <ProcedimientoRealizado>
                <Descripcion>Apendicectomia</Descripcion>
                <NombreProfesional>Dra. Isabela Ramisoni</NombreProfesional>
                <Fecha>3/10/1995</Fecha>
            </ProcedimientoRealizado>
        </ProcedimientosRealizados>
        <FarmacosEnUso>
            <Farmaco>
                <Nombre>Aspirina</Nombre>
                <Dosis>
                    <Unidades>1</Unidades>
                    <Via>Oral</Via>
                    <Intervalo>8 horas</Intervalo>
                </Dosis>
                <Desde>2/2/2001</Desde>
                <TiempoUtilizacion>1 semana</TiempoUtilizacion>
            </Farmaco>
            <Farmaco>
                <Nombre>Zolben</Nombre>
                <Dosis>
                    <Unidades>2</Unidades>
                    <Via>Oral</Via>
                    <Intervalo>24 horas</Intervalo>
                </Dosis>
                <Desde>9/2/2001</Desde>
                <TiempoUtilizacion>2 semanas</TiempoUtilizacion>
            </Farmaco>
        </FarmacosEnUso>
</HistoriaClinica>
```

aiii.5. Código relevante

A continuación se presenta el código fuente de las principales clases implementadas para la solución propuesta del caso de estudio. Estas son:

- **TSystem.java**: Clase sistema de la solución a nivel del CAD. Esta combina la capa lógica y la capa de persistencia del CAD para brindar las siguientes funcionalidades: cargar un árbol de la tarjeta, guardar un árbol en la tarjeta, cargar un árbol desde un archivo SML, guardar un árbol a un archivo SML, cargar un template desde un archivo SML y guardar un template a un archivo SML
- **Storage**: Clase principal de la capa de persistencia en el CAD. Como se presentó en el punto 4.3.5
- **MemoryManager**: Esta clase implementa representa el manejador de memoria implementado dentro de la tarjeta. Como se mencionó en el punto 4.3.4

TSystem.java

```
package t5.treedata;


import t5.sccad.*;
import java.io.*;


public class TSystem {

  private SMLTemplate lvTemplate = null;

  private Tree lvTree = null;


  public TSystem() {

  }


  public void loadTemplate(String pFileName, String pRootNodeName) {
    lvTemplate = new SMLTemplate(pRootNodeName);
    lvTemplate.loadTemplate(pFileName);

    lvTree = null;

  }

  public Tree getTemplateTree() {
    return lvTemplate.getTree();

  }

  public String getRootNodeName() {
    return lvTemplate.getRootNodeName();

  }

  public void storeTemplate(String pFileName) throws IOException {
    lvTemplate.storeTemplate(pFileName);

  }









  public void loadFromCard(AID pAID) {

    CardStorage card = new CardStorage(pAID);

    load(card);

  }
```

```
  public void storeInCard(AID pAID) {

  CardStorage card = new CardStorage(pAID);

    store(card);

  }

  public void loadFromFile(String pFileName) {

    SMLStorage stg = new SMLStorage();
    stg.setRootName(getRootNodeName());
    stg.setInputFileName(pFileName);

    load(stg);

  }

  public void storeInFile(String pFileName) {

    SMLStorage stg = new SMLStorage();
    stg.setRootName(getRootNodeName());
    stg.setOutputFileName(pFileName);

    store(stg);

  }


  private void load(Storage pStorage) {
    try {
      lvTree = pStorage.loadTree(getTemplateTree());

    }
    catch (Exception e) {
      System.out.println(e.getMessage());
      e.printStackTrace();

    }

  }

  private void store(Storage pStorage) {

    pStorage.storeTree(lvTree);

  }

}
```

Storage.java

```
package t5.treedata;

import java.lang.Class;
import java.util.*;
import t5.sccad.*;


public abstract class Storage {

  /**
  *  This method travers the template, create the info of the node from
  *  from the type defined in the template and call the method loadInfo
  *  for each node of the template tree.
  *
  *  The type of the info, of the node's, of the template tree is
  *  InfoString. And the data inside of the info is the name of the
  *  class that the info of that node belong to.
  */
  public final Tree loadTree(Tree template) throws ClassNotFoundException,
InstantiationException, IllegalAccessException
  {
    Tree newTree;
    Node root;
    Node tmp;
    Vector idPath = new Vector(1,5);

    beginLoad();

    // Create the root node
    root = new Node(new String(template.getRoot().getName()));

    // Create the new tree
    newTree = new Tree(root);

    for(int i = 0; i < template.getRoot().countChildren(); i++)
    {
      idPath.add(new Integer(i + 1));

      tmp = loadTree_util(template.getRoot().getChildAt(i), idPath,
template.getRoot().getChildAt(i).getName());

      root.addChild(tmp);

      idPath.remove(0);
    }

    endLoad();

    return newTree;
  }

  private Node loadTree_util(Node template, Vector idPath, String path)
throws ClassNotFoundException, InstantiationException,
IllegalAccessException
  {
    Node newNode;
    Node tmp;
    int countElems = 0;

    if (template.isLeaf())
    {
      // If the node info exists, it create an instance of
      // that class (the data in the info of the node,
      // is the name of the class to create).
      // Copy the structure of the node
      if (template.getInfo() != null){
        newNode = new Node(new String(template.getName()),
generateInfo(template.getInfo().toString(),idPath,path));
      }else{
        newNode = new Node(new String(template.getName()), null);

      }

      return newNode;
```

```
    }else
    {
      if (!template.getisCollection())
      {
        if (template.getInfo() != null){
          newNode = new Node(new String(template.getName()),
generateInfo(template.getInfo().toString(),idPath,path));
        }else{
          newNode = new Node(new String(template.getName()), null);
        }

        for(int i = 0; i < template.countChildren(); i++)
        {
          idPath.add(new Integer(i + 1));

          tmp = loadTree_util(template.getChildAt(i), idPath, path + "." +
template.getChildAt(i).getName());

          newNode.addChild(tmp);

          idPath.remove(idPath.size() - 1);
        }
        return newNode;
      }
      else
      {
        // Create the collection node.
        if (template.getInfo() != null){
          newNode = new Node(new
String(template.getName()),generateInfo(template.getInfo().toString(),idPath
,path),true);
        }else{
          newNode = new Node(new String(template.getName()),null,true);
        }

        // Get the number of elements of the collection
        try{
          countElems = getNumberChildren(path, convertArrayInt(idPath));
        }catch(Exception e){
          System.out.println("Storage.loadTree_util: Error al obtener
numero de hijos de colección");
        }


        for (int i = 1; i <= countElems; i++)
        {
          // Add the first index of the idPath
          idPath.add(new Integer(i));

          tmp = loadTree_util(template.getChildAt(0), idPath, path + "(" +
new Integer(i).toString() + ")." + template.getChildAt(0).getName());

          newNode.addChild(tmp);

          idPath.remove(idPath.size() - 1);
        }
        return newNode;
      }
    }
  }

  /**
  *  Convert an array of vector of Integers to an array
  *  of bytes.
  */
  private int[] convertArrayInt(Vector arr){
    int[] arrInt;
    int i;

    if(arr.size() != 0){
      arrInt = new int[arr.size()];
    }else{
      return null;
    }

    for(i = 0;i < arr.size();i++){
```

```
        arrInt[i] = ((Integer) arr.elementAt(i)).intValue();
      }


      return arrInt;
    }

    /**
    *  This method recives the name of a class, create an instance
    *  of that class and load the info of the node (calling the
    *  loadInfo method).
    */
    private Info generateInfo(String className, Vector idPath, String path)
throws ClassNotFoundException, InstantiationException,
IllegalAccessException{
      Info tmp;

      tmp = (Info) Class.forName("t5.treedata." +  className).newInstance();

      try{
        loadInfo(path, convertArrayInt(idPath), tmp);
      }
      catch(InfoNotFoundException e) {
        tmp = null;
      }
      catch(Exception e) {
        System.out.println("Error al buscar la información.");
      }

      return tmp;
    }


    public final void storeTree(Tree treeOfData){
      Node root;
      Node tmp;
      Vector idPath = new Vector(1,5);

      beginStore();

      for(int i = 0; i < treeOfData.getRoot().countChildren(); i++)
      {
        // Add the first index to the idPath
        idPath.add(new Integer(i + 1));

        storeTree_util(treeOfData.getRoot().getChildAt(i), idPath,
treeOfData.getRoot().getChildAt(i).getName());

        // Delete the element at the position 0
        idPath.remove(0);
      }

      endStore();

    }

    private void storeTree_util(Node nodeOfTree, Vector idPath, String path)
    {
      Node newNode;
      Node tmp;

      if (nodeOfTree.isLeaf())
      {
        if (nodeOfTree.getInfo() != null){
          // Store the info in the card, SML file or other media.
          try{
            storeInfo(path, convertArrayInt(idPath), nodeOfTree.getInfo(),
nodeOfTree.getUpdated());
          }catch(Exception e)
          {}
        }
      }else
      {
        // Store the info in the card, SML file or other media.
        if (nodeOfTree.getInfo() != null){
          try{
```

```
            storeInfo(path, convertArrayInt(idPath), nodeOfTree.getInfo(),
nodeOfTree.getUpdated());
          }catch(Exception e){}
        }

        for(int i = 0; i < nodeOfTree.countChildren(); i++)
        {
          // Add the first index to the idPath
          idPath.add(new Integer(i + 1));

          // Call the method recursivly to travers the tree
          if (!nodeOfTree.getisCollection()){
            storeTree_util(nodeOfTree.getChildAt(i), idPath, path + "." +
nodeOfTree.getChildAt(i).getName());
          }else{
            storeTree_util(nodeOfTree.getChildAt(i), idPath, path + "(" + new
Integer(i+1).toString() + ")" + "." + nodeOfTree.getChildAt(i).getName());
          }

          idPath.remove(idPath.size() - 1);
        }
      }
    }
    protected abstract void beginLoad();

    protected abstract void loadInfo(String path, int[] idPath, Info theInfo)
throws NodeNotFoundException, InfoNotFoundException;

    protected abstract int getNumberChildren(String path, int[] idPath) throws
NodeNotFoundException;

    protected abstract void endLoad();

    protected abstract void beginStore();

    protected abstract void storeInfo(String path, int[] idPath, Info theInfo,
boolean isDirty);

    protected abstract void endStore();
}
```

MemoryManager.java

```
package t5.prototipo;

public class memoryManager{

  // Instance of the table. Keep record of the data stored.
  private table lvTable;

  // Instance of the Dinamic Memory. Keep the data stored.
  private dinMem lvMemory;

  // Instance of the sequence. It's actualy represents the memory where
  // the table and the dinamic memory is stored.
  private sequence lvSequence;

  public memoryManager(short size, dataPackage dataIn, dataPackage dataOut){

    lvSequence = new sequence(size, dataOut);

    lvTable = new table(lvSequence, dataIn);
    lvMemory = new dinMem(lvSequence);
  }

  /**
   *  Create a new information with <i>id</i> as its identifier
   *  and <i>data</i> as its asociated data.
   *
   *  @param id Node path
   *  @param data Data asociated with the node
   *
   */
  public void addInfo(idPath id,dataPackage data) {
    short offset;
```

```
    offset = lvMemory.addBlock(data);
    lvTable.addElement(id,offset,data.length);
  }

  /**
   *  Remove the information identified by <i>id</i> from the memory.
   *
   *  @param id Node identifier
   *
   */
  public void removeInfo(idPath id){
    // Remove the element from the table
    // There is no need of removeing the element from the
    // memory.
    lvTable.removeElement(id);
  }

  /**
   *  Update the data of the information identified by <i>id</i>.
   *
   *  @param id   Node identifier
   *  @param data Data asosiated with the node
   *
   */
  public void updateInfo(idPath id, dataPackage data){
    short offset;
    short length;
    short newOffset;

    offset = lvTable.getElementOffset(id);
    length = lvTable.getElementLength(id);

    newOffset = lvMemory.updateBlock(offset,length,data);

    lvTable.updateElement(id, newOffset, data.length);
  }

  /**
   *  Return the number of children, of the block identified <i>id</i>.
   *  Using the codification used to the information
   *
   *  @param id Father node identifier
   *
   *  Note: The maxium number of children is 256 (0..255)
   *
   */
  public byte numberOfChildren(idPath id){
    return lvTable.numberOfElemetsWithPrefix(id);
  }

  /**
   *  Return the data of the information identified by <i>id</i>.
   *
   *  @param id Node identifier
   *
   */
  public dataPackage getInfo(idPath id){
    short offset;
    short length;

    offset = lvTable.getElementOffset(id);
    length = lvTable.getElementLength(id);

    return lvMemory.getBlock(offset,length);
  }

  /**
   *  Return <i>true</i> if the information identified by <i>id</i>
   *  the collection and <i>false</i> otherwise.
   *
   *  @param id Node identifier
   *
   */
  public boolean member(idPath id){
    return lvTable.member(id);
  }
```

```
   /**
    *  Return the free space avalable.
    *
    */
   public short getStoregeFreeSpace(){
     return lvSequence.freeSpace();
   }
}
```

Apéndice IV
SML

aiv.1. Introducción

XML (Extensible Markup Language) es una especificación desarrollada por W3C [W3C]. XML es una versión reducida de SGML (Standard Generalized Markup Language), diseñada para representar información en forma estructurada. Permite a los desarrolladores diseñar su propio juego de marcas (tags) para satisfacer requerimientos particulares, permitiendo la definición, transmisión, validación e interpretación de datos entre aplicaciones y entre organizaciones.

SML (Simple Markup Language) es un subset de XML. Puede considerarse a SML como una simplificación de XML que no presenta atributos, instrucciones de procesamiento, y declaraciones de tipo de documento (DTD), entre otras características [SML1].

Surge de la experiencia de desarrolladores y organizaciones que utilizan XML en sus aplicaciones. Los mismos notaron que gran parte del potencial de la especificación XML 1.0 de W3C no estaba siendo utilizado. Varios foros de discusión tratan la contingencia entre XML y SML, siendo un tema de gran actividad desde finales de 1999 [SML2].

aiv.2. Especificación

No se cuenta con una especificación estándar de SML. Se han propuesto varias versiones de la gramática de SML, la cual ha sido el mecanismo de obtención de feedback en los foros de discusión, para llevar adelante a SML.

A continuación se presenta una versión preliminar de la gramática de SML, que fue publicada en noviembre de 1999 [SML3]

```
data ::= (element | CharData | CharCode)*
element ::= StartTag data EndTag | EmptyTag
StartTag ::= '<' Name '>'
EndTag ::= '</' Name '>'
EmptyTag ::= '<' Name '/>'
Name ::= NameChar+
NameChar ::= [^<>&] - (#x20 | #x9 | #xD | #xA)
CharData ::= [^<>&]*
CharCode ::= '&' ('#x' [0-9A-F]+ | 'lt' | 'gt' | 'amp') ';'
```

aiv.3. Utilización de SML

En el desarrollo del prototipo de optó por la utilización de SML principalmente por su simplicidad, aunque también por la legibilidad que presentan los documentos de este tipo.

Se desarrolló un parser de SML utilizando las herramientas Jlex y Cup. La gramática utilizada se presenta a continuación:

```
digit      ::= [0-9]

letter     ::= [a-zA-Z]

identifier ::= letter (letter | digit)*

info       ::= [0-9a-zA-Z.,()$%"#]

strinfo    ::= INFO
             | INFO strinfo

tag        ::= '<' identifier '>'

coltag     ::= '<' identifier '*>'

closetag   ::= '</' identifier '>'

sml        ::= TAG strinfo CLOSETAG
             | TAG content CLOSETAG
             | COLTAG content CLOSETAG

content    ::= sml
             | sml content
```

La gramática propuesta permite documentos con las siguientes características:

- Un tag con su correspondiente closetag deben coincidir en el identificador
- Todo se encuentra delimitado por un tag y su closetag correspondiente
- Los tags que indican que un nodo es una colección deben ser seguidos de al menos una pareja de tag y closetag. Además, el closetag correspondiente no debe llevar '*'. La extensión de coltag se realizó para utilizarlo en el template.
- Entre un tag y su correspondiente closetag puede hallarse texto o una lista de tag con su correspondiente closetag, pero no ambos.